6-25-2015

Usage Management Enforcement in Cloud Computing Virtual Machines

Edward J. Nava

Edward J. Nava
Candidate

Electrical and Computer Engineering
Department

This dissertation is approved, and it is acceptable in quality and form for publication:

Approved by the Dissertation Committee:

Gregory L. Heileman, Chairperson

James Plusquellic

Jed Crandall

Marios Pattichis

Edward D. Graham, Jr.

USAGE MANAGEMENT ENFORCEMENT IN CLOUD COMPUTING VIRTUAL MACHINES

by

EDWARD J. NAVA

Associate of Science, Electronic Engineering Technology, New Mexico State University, 1973
Bachelor of Science, Electrical Engineering, University of New Mexico, 1979
Master of Science, Electrical Engineering, Stanford University, 1980

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Engineering

The University of New Mexico
Albuquerque, New Mexico

May, 2015

DEDICATION

This is dedicated to my wife Jeanette and my entire extended family. They have provided continued support and encouragement throughout this effort. They have also been very understanding that I spend large amounts of time on academics.

ACKNOWLEDGEMENTS

I wish to thank Dr. Greg Heileman for his encouragement and guidance through the classroom and research activities. His can do attitude has been an inspiration to all.

I also wish to thank the Informatics Research Team, including Dr. Chris Lamb, Dr. Juan-Marcio Luna, and Viswanath Nandina, for their collaborations in cloud computing research. The team has helped motivate me to pursue the research topic of developing a usage management enforcement mechanism for use in cloud computing.

I also wish to thank my committee members: Dr. Jim Plusquellic, Dr. Marios Pattichis, Dr. Jed Crandall, and especially Dr. Ed Graham, who has provided very helpful reviews and continuous encouragement.

Usage Management Enforcement in Cloud Computing Virtual Machines

By

Edward J. Nava

Associate of Science, Electronic Engineering Technology

Bachelor of Science, Electrical Engineering

Master of Science, Electrical Engineering

Doctor of Philosophy, Engineering

ABSTRACT

Many are interested in adopting cloud computing technology, but have concerns about the security of their data. This issue has motivated extensive research to address potential vulnerabilities, with a major focus on access control. A related cloud computing concern is controlling what users can do with data to which they have been granted access. This control is needed to prevent accidental loss or deliberate theft of data by users who have been granted legitimate access. The need for this control, called usage management, has led to a number of conceptual approaches for both conventional and cloud computing, all of which will require an enforcement mechanism within the processor's domain. The goal of this research is to prove that it is possible to implement a completely software-based enforcement mechanism that can operate independently of the application software. The implementation is based on a formal operational model. A number of implementation approaches were considered in formulating the

enforcement strategy. Then, leveraging software instrumentation capabilities and extending tools developed for taint analysis, we developed a software-based usage management enforcement mechanism that uses dynamic data flow tracking. Based on usage flow policies that are specified in machine readable licenses, the enforcement mechanism can permit or inhibit data flows to standard interfaces, data files, and network sockets. The enforcement mechanism does not require direct hardware access, so it can be used very effectively in a cloud computing environment. This demonstrated capability now provides information owners an ability to control what authorized users can do with the information.

Table of Contents

List of Figures

List of Tables

Chapter 1 – Introduction

Computing environments have evolved significantly over the last fifty years. Early computers executed programs in a sequential mode, where the entire system was devoted to executing one single program at a time. With this batch mode, there were no concerns about information inadvertently flowing from one program to another as the system could be powered down between jobs to eliminate any data remnants. Processing data of varying sensitivity levels was straightforward, though turn-around time could be lengthy.

Later, time-shared systems were developed to provide multiple users an ability to simultaneously access the machine for software development and timely execution of programs. The multiprocessing operating systems provided an environment where users had access to all of the machine resources, though they were shared with other users. The storage devices were capable of storing multiple users' files, so the systems included capabilities for *users* to designate who could access their files, though privileged users could override these access controls. This type of Discretionary Access Control (*DAC*) capability is still used in today's operating systems.

The early computers were very expensive, so there was a high level of interest in being able to use them to process information of different sensitivity levels simultaneously. Processing data of different sensitivity levels would require rigorous controls for data access and transmission. These needs inspired much of the initial research on Multi-Level Security Systems, Flow Control, and Covert Channels.

As computer hardware technology advanced and procurement costs decreased, some organizations chose to use isolated systems for processing sensitive information. Economical personal computer technology was another catalyst for the use of isolated systems. Information brought into an isolated system might be read-in using removable media, which would then be destroyed or thereafter be handled as data of the same sensitivity level as the target machine. Any data removed from the high level system would use a rigorous human review process to ensure that no sensitive data was being extracted inadvertently.

Network communications have greatly impacted computing environments as now vast quantities of information can quickly be accessed and shared. Organizations processing sensitive data have adopted network technology to improve capabilities and effectiveness, but these systems have usually been configured to maintain complete isolation from public networks.

A large fraction of historical network usage has been based on a client-server model, where a user's machine operates as a client that regularly requests information from servers and occasionally, sends a significantly smaller amount of data back to the server. Today, web applications are evolving from simple content servers and provide much more functionality including data storage, web-hosted email services, and other applications that were formerly executed on client machines. Many of these web applications are operated by companies such as Google, Amazon, and Microsoft and as they have built up computing facilities to host these web applications, they and others have

developed extensive computing enterprises that they now rent portions of to other users. In this mode of operation, the users are physically removed from the computing resources that are executing their applications and from their perspective; the supporting resources are figuratively located in the “clouds”.

Many organizations are adopting cloud computing in order to reap the benefits of being able to quickly and economically establish an extensible computing enterprise. Other organizations are also interested in the technology, but are hesitant to adopt it because of concerns over the security of their information. These concerns provide the motivation for extensive research regarding information security in cloud computing systems.

A major objective of information security is to provide information owners an assured ability to control who has access to their data. Usage Management (*UM*) compliments access control by providing an ability to control what a user can do with data once they have been legitimately granted access (Park & Sandhu, 2004) (Jamkhedkar, Heileman, & Lamb, 2010). In other words, a comprehensive UM system includes both an access control and continuous policy-based enforcement capability.

Usage Management has some common objectives with Digital Rights Management (DRM). A DRM system manages the appropriate use of digital content and its objective is to prevent the illegal use of licensed content; the primariy motivation for using DRM is to prevent loss of revenue (Subramanya & Yi, 2006) (Liu, Safavi-Naini, & Sheppard, 2003).

DRM uses licenses, which are separate from the content whose use is being controlled. The content may be encrypted or encoded in a proprietary format that is suitable for tracking and management of its usage; the content cannot be used without a valid license. So, in order to use the content, the consumer must purchase a license granting usage rights and these rights are often tied to a particular client machine. Consumer devices which use the content must be able to properly interpret the usage rules specified in the license. Applications which play DRM protected content must be augmented with plug-ins by the DRM provider in order to access the digital content. Digital content protected by one DRM system cannot be accessed by the client-side application in another DRM system, so applications may need multiple plug-ins. Also, some DRM systems rely on hardware to both identify the client machine as well as implement cryptographic functions needed to access the content. The need to ensure that applications are extended with DRM capabilities and potential dependence on hardware interactions limits the use of DRM. In view of the issues with DRM, a usage management capability which does not require hardware access or cooperative applications is essential for use in cloud computing.

Cloud computing is a broad term, which includes various service models. This research is focused on the Infrastructure as a Service model (*IaaS*) (Liu, et al., 2011) (Badger, Grance, Patt-Comer, & Voas, 2011). With the IaaS model, the cloud subscriber controls both the operating system and application software executing in Virtual Machines (*VMs*) and the cloud computing provider has

control of the hardware and the Virtual Machine Manager (*VMM*) (also known as *hypervisors*) that hosts the user's VMs, as shown in Figure 1.

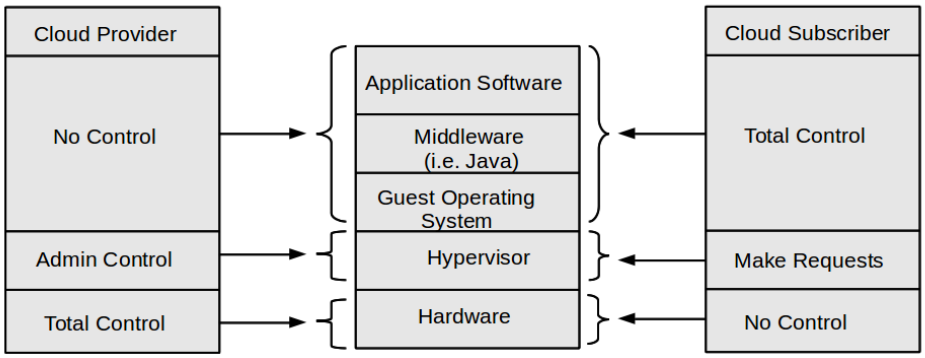

Figure 1 Infrastructure-As-A-Service (IAAS) Control

With IaaS, the ability to instantiate VMs is a key function. A cloud user can configure the VMs, which provides an opportunity to add mechanisms for information security or usage management. As the cloud user may not have control of, or access to, the actual hardware or software hosting their VMs, software executing within the VM must provide the desired protections and control mechanisms.

Computing systems incorporate a hierarchical design as shown in Figure 2. The operating system manages all system resources and provides the environment in which the application software executes. When the application software requires access to any system resources, it uses a library function call, such as *printf*(), which in turn results in an operating system call to access the desired resource. The operating system uses device driver software to access

hardware resources. There are two logical interfaces where additional control functionality can be added: between the application layer and the operating system, and between the operating system and the hardware. VMMs, use the latter interface to isolate operating system access to the hardware. This hierarchy can be leveraged for enhancing information security or for system exploitation. This hierarchy can also be used for usage management enforcement mechanisms.

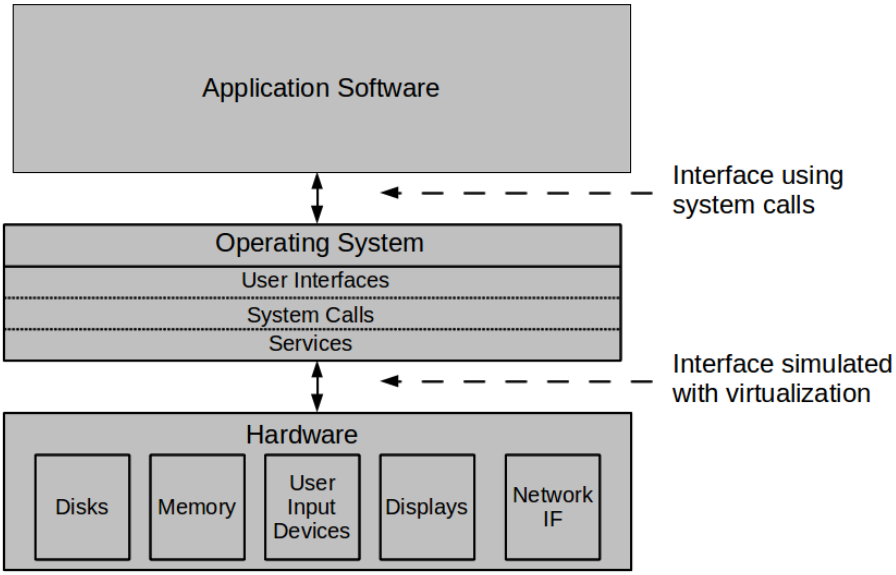

Figure 2 Hierarchical Computation Configuration

As consumers, many of us interact with *public* cloud computing services provided by commercial entities such as: Amazon, Google, Apple, and Microsoft. However, some organizations choose to implement cloud computing systems using their own resources and they are called *private* clouds. For example, the Department of Defense has a private system which is called m*ilCloud*. With a

private cloud, the owning organization can have consolidated systems that are easier to control and administer, and it has complete control of the entire hardware and software suite. This complete control provides a greater ability to protect its information. The downside is that it must make the financial investment to buy and operate the system. For this reason, "renting" computing resources from a public cloud computing provider is very attractive; the downside is lack of control.

As noted, the UM concept includes the ability to control how a user, who has been granted legitimate access, uses the data. Existing models that describe UM operations assume the existence of an enforcement mechanism within the target processor. However, no one has proposed a specific approach for implementing the enforcement mechanism. The fundamental objective of this research is to answer the questions*:*

It is possible, using a completely software-based approach, to implement a usage management enforcement mechanism within a specific processor environment, and in particular, a virtual one that is executing in a public cloud computing environment? Can a usage management capability be implemented without modifying the applications software? Can the capability be implemented with no changes to the guest operating system?

To illustrate why a usage management enforcement capability is needed, consider the following real-life scenario: In 2014, the Russian Federation hosted the Winter Olympics in Sochi (Olympics.org, 2014). Prior to the Olympics, there were concerns about possible terrorist activities intended to disrupt the Olympics.

In addition, there have been ongoing political tensions amongst the participating countries, but circumstances required their law enforcement and intelligence communities to share information in order to assure the safety of the athletes and spectators during the event. The countries providing information likely wanted to limit how broadly their sensitive information was shared by host country analysts using the data. For this kind of situation, having a community cloud computing environment that includes an automatic means of controlling how data are used would be very desirable.

Usage management can also be very beneficial in environments where organizations seek to limit how their employees download data from their work computer systems. There are numerous examples of where employees have downloaded sensitive data to their laptop computers and then lost the laptops, or had them stolen. Being able to prevent these unwanted downloads could also be very desirable.

In this dissertation, a formal description of the usage management enforcement mechanism operation is provided. This is followed by a review of a set of existing technical capabilities that are routinely used to secure information and an assessment of their applicability to this problem. Next, related research and highlight implementation strategies that may lend themselves to usage management enforcement are presented. This is followed by a discussion of the experimental configurations that were used to test the proposed UM enforcement mechanism with a summary of the results and their significance. Finally, the

effectiveness of this approach is assessed and areas of future research that can augment the demonstrated capability are identified.

Chapter 2 – An Operational Model for Usage Management

Before proposing a solution, it is necessary to provide a comprehensive description of what the usage management enforcement mechanism is supposed to do. For this purpose, historical information security research is very appropriate, as the fundamental problems have not changed much since the early 1970's.

Information security concerns have existed since the time when multiple users shared access to central computing systems. There was a need to process information of different sensitivity levels on these machines and there was also a requirement to ensure that users could only access information for which they were authorized. In this type of environment, non-sensitive data can be accessed by all users; as the sensitivity level increases, the set of users who can assess the data is increasingly restricted. This type of operation is called a *Multi-Level Security (MLS)* system. Early MLS implementation needs motivated considerable research, and one of the first security models was proposed by Bell and LaPadula (Bell & LaPadula, 1973) (LaPadula & Bell, 1996). We can relate this model to usage management and the actions of the enforcement mechanism.

The Bell LaPadula model has three components which the authors call: (1) the simple security principle, (2) the * principle, and (3) the tranquility principle. The three principles can be described both in general terms and in a formal mathematical way. The simple security principle means that a user is prohibited from *reading* all information that is at a higher sensitivity level than for

which he or she is authorized. The * principle restricts a user from *writing* information to any level below that for which he or she has active object access. The tranquility principle states that the security classification of active objects will not be changed during normal operations. The first two principles (sometimes called properties by some authors) are often stated as "no-reads-up" and "no-writes-down" (Bishop, 2003). (For the rest of this document, the three components are referred to as properties.)

An MLS system is typically uses security levels and compartments within each level. For simplicity, assume that all information is in one compartment. Higher security levels imply higher levels of sensitivity, so it is mandatory that information from the higher levels not flow to the lower levels.

Subjects, or users, are approved for access by sensitivity levels. However, authorized access to information at a given level does mean that a subject can access all information at that level. In a military security model, the access restrictions within a level are called *need to know*. While level authorization is a necessary condition for access of an object a given level, it is not a sufficient condition on which to grant access.

The Bell LaPadula model is a lattice-based Mandatory Access Control (*MAC*) system which references an access control matrix entry to determine if a subject, *S*, is allowed to access an object, *O*. Because it is a MAC system, the user (subject) is not allowed to change the security attributes of any object.

As noted, a comprehensive UM system includes both an access control capability and the means to enforce usage policies while the object is in use. In

a cloud computing environment with a hierarchical UM system, a centralized access control function is implemented outside of the VM and is responsible for implementing the simple security property. The UM enforcement mechanism within the VM must enforce the * property and the tranquility property; it must be capable of preventing the flow of information from objects of higher sensitivity categories to objects of lower sensitivities.

To illustrate more formally, assume that we have a data set, *A*, at a high sensitivity level and another, *B*, at a lower level. Then we say that *A dominates B*, or in a shorthand notation $A\ dom\ B$. If $x \in A$ and $y \in B$, then an assignment statement which is moving data from x to y in an executing program, i.e. $y = x$, would not be allowed, as it would violate the * property. To enforce this constraint, a system must detect flows of information from the higher sensitivity levels to the lower sensitivity levels. In contrast, an assignment instruction of the form $x = y$ would be allowed, as it is in compliance with the simple security property, i.e. "read down".

The two assignment statements shown in the previous paragraph represent simple examples of explicit information flow. Assignment statements can include other modifiers such as: *AND, OR, XOR, ADD, SUB, MUL, DIV*, etc. that while modifying the original data, still constitute explicit data flows. Another class of instructions, which change execution paths based on the values of x, may result in implicit data flows that are also a concern. In this research, the focus will be on explicit data flows.

Recognition of the information flow problem in MLS systems led to extensive research to detect and control the flow of information. Bishop describes the properties of information flow and some of the approaches which have been proposed to detect and prevent unwanted information flow. An information flow policy can be expressed as a triple, $I = (SC_I, \leq_I, join_I)$, where SC_I is a set of security classes, $\leq_I$ is an ordering relation, such as dom, and $join_I$ combines two elements of SC_I.

A variation of Foley's confinement flow model can be used to illustrate the operation of a UM mechanism within a VM (Foley, 1989). The confinement flow model is a 4-tuple $(I, O, confine, \rightarrow)$, in which $I = (SC_I, \leq, join_I)$ is a lattice-based information flow policy; O is a set of entities; $\rightarrow: O \times O$ is a relation with $(a, b) \in \rightarrow$ if and only if information can flow from a to b; and for each $a \in O$, $confine\ (a)$ is a pair $(a_L, a_U) \in SC_I \times SC_I$, with $a_L \leq a_U$. What this means is that for $a \in O$, if $x \leq a_U$, then information can flow from $x\ to\ a$ and if $a_L \leq x$, information can flow from $a\ to\ x$. Therefore, if information can flow from $a\ to\ b$, then $b\ dom\ a$ and this becomes:

$$(\forall\ a, b \in O)[a \rightarrow b \Rightarrow a_L \leq_I b_U] \qquad (1)$$

In Foley's model, there is an assumption that an object can change security classification, which is contrary to the Bell LaPadula tranquility property. For this research effort, the object security classifications are fixed. If data or user security classifications change, the centralized UM controller can terminate the execution in the VM, if necessary.

The operation that is shown in equation 1 represents the fundamental action that a UM enforcement mechanism within a VM must perform. Sensitive information to be controlled dominates all non-sensitive information; flows from the sensitive to non-sensitive must be prevented.

Chapter 3 – Potential Solutions Using Existing Technical Capabilities

Before proposing a solution for enforcing usage management within a VM, it is useful to examine whether existing technologies can be used to accomplish the objective of preventing a user from misusing the data to which he or she has been granted access. This chapter presents several technologies that are currently in use and discusses why they will not achieve the desired goals.

Information usually has value to the owner. The consequence of loss may range from personal embarrassment or personal privacy concerns, to significant financial or national security consequences. The value generally determines the extent of the measures that information owners are willing to use to protect it. In the case of high-value information, the importance of protecting it has justified the large expense of completely segregating the information processing systems or establishing extensive infrastructure to support security functions. It may also justify a significant amount of processing overhead to provide ongoing protection of data. For many organizations, operating completely isolated systems is not a viable approach, leading to the use of the protection techniques described here.

The widespread availability of high speed computer network communications has made cloud computing a very attractive option, but the information security concerns are even greater as the users no longer have physical control of the computing resources. However, the technologies described here can be used to provide users assurance that data at rest and data in transit are secure. Extensions of these technologies may also be

considered for protecting data in use in cloud computing VMs. They are described below.

3.1 Encryption

When thinking about information security, one approach that immediately comes to mind is encryption (Schneier, 1996). If data are stored in an encrypted form, then, assuming that the encryption is highly resistant to attack, there is less concern if remnants of the data remain after a delete operation. Similarly, encrypted data is generally safe when being transferred from a cloud storage repository to a virtual machine for processing. Depending on the application, the data may need to be decrypted for effective use within a VM.

When considering using encryption, one must consider the implementation details carefully to ensure that the encryption does not provide a false sense of security. Let us consider some of the issues. First, using an encryption algorithm, E, and an encryption key, KE, we generate a ciphertext, C, that is an encrpyted version of the original message, M. The encryption process is represented by the following formula:

$$\mathrm{C} = \mathrm{E}_{\mathrm{KE}}(\mathrm{M}) \qquad (2)$$

To use the data for subsequent use, the data must be decrypted using a decryption algorithm, D, and a decryption key, KD, to recover the message, M.

$$\mathrm{M} = \mathrm{D}_{\mathrm{KD}}(\mathrm{C}) \qquad (3)$$

Generally, the algorithms for encryption and decryption are publicly known and the secrecy is associated with the keys. With *symmetric* encryption algorithms, the same key is used for both encryption and decryption, i.e. $KE =$

KD. Keeping the key secret is essential for ensuring that the ciphertext will be secure; there is a significant challenge in securely distributing the key to all parties engaged in the communication of the data. Generally, keys are distributed through a means other than the data communication channel, such as paper tapes, code books, and electronic storage devices. If many users communicate using the same key, the potential for compromise increases. For example, in the 1980s, the US Navy experienced very significant fleet-wide security compromises when the secret keys were compromised by John Walker (Richelson, 1995).

An alternative approach is to use an *asymmetric* encryption algorithm, where two different keys are used for the encryption and decryption processes. Users have both a *public key,* KE, and a *private key,* KD. A sender encrypts a message using the receiver's public key. The message can then only be decrypted using the receiver's private key. As with symmetric algorithm encryption, asymmetric algorithm encryption has key security challenges, as well; the user's private keys must be generated or distributed securely, and measures are needed to assure that a published public key actually belongs to the intended recipient.

Both symmetric and asymmetric encryption algorithms are in widespread use today to protect information from unauthorized access. Because of processing efficiencies, asymmetric encryption is typically used in the initial stages of a prolonged communication session, to exchange an encrypted symmetric key; subsequent communication in the session is done using

symmetric encryption. Encryption is also used to secure data in storage, often using hardware for part of the implementation.

3.2 Homomorphic Encryption

RSA is an example of an asymmetric key algorithm that, in its basic form, has some weaknesses (Paar & Pelzl, 2010). One specific property of interest for this discussion is that it is *malleable*. To put the characteristic in perspective, first consider the encryption and decryption algorithms, where KE, KD, and n are derived in a key generation process not shown here. The encryption and decryption algorithms use exponentiation and the modulus functions as shown below:

$$\mathrm{C} = \mathrm{M}^{\mathrm{KE}} \bmod \mathrm{n} \tag{4}$$

$$\mathrm{M} = \mathrm{C}^{\mathrm{KD}} \bmod \mathrm{n} \tag{5}$$

If the attacker replaces the ciphertext C with $S^{KE}C$, where S is some integer, then when the receiver decrypts the modified ciphertext, he gets:

$$(\mathrm{S}^{\mathrm{KE}}\mathrm{C})^{\mathrm{KD}} = \mathrm{S}^{\mathrm{KE\,KD}}\mathrm{M}^{\mathrm{KE\,KD}} = \mathrm{S\,M} \bmod \mathrm{n} \tag{6}$$

While an attacker does not get access to the original message, he is able to modify it in a way that could be harmful. The malleability property illustrates the characteristic that is the basis for *homomorphic* encryption, where an ability to modify ciphertext in a predictable way may be useful.

Gentry proposed a fully homomorphic encryption method that allows any efficiently computable function, f, to be applied to encrypted data so that a user can manipulate data in a useful way without ever actually having access to the unencrypted data (Gentry, 2010). This is done using an *Evaluate* algorithm

which generates a modified cipher text. Here, C' represents an encrypted version of $f(M)$ and C represents the encrypted version of M. The process is illustrated with the following equation:

$$\mathrm{C}' = \mathrm{Evaluate}_{\mathrm{KE}}(\mathrm{f}, \mathrm{C}) \quad (7)$$

Fully homomorphic encryption could be attractive for some cloud computing applications. Encrypted data could be downloaded into a virtual machine, processed using a set of arbitrary functions, and then stored back in the data storage repository. There are concerns on the practicality of homomorphic encryption; Lauter et al. show that schemes that are limited to a small number of functions can be much faster than fully homomorphic schemes, and can indeed be practical (Lauter, Naehrig, & Vaikuntanathan, 2011). There are limits to the usefulness of homomorphic encryption because in most situations, a user will actually need access to the unencrypted data, so alternative approaches are necessary.

3.3 Trusted Platform Module

Any software that is used to provide a means of information security is vulnerable to software attack. The Trusted Platform Module, *TPM,* is a separate hardware microcontroller that can securely store keys, certificates, and signatures. It includes a math-coprocessor that implements cryptographic operations such as asymmetric key generation, asymmetric algorithm encryption, hashing, and random number generation (TCG, 2011)**.** The TPM standard was developed by the Trusted Computing Group, but the actual devices are manufactured by industrial companies and have been integrated into most

modern laptop, desktop, and server computers. The TPM has been used by Microsoft in its implementation of BitLocker®. In private cloud computing systems, the owning organization has complete control over the system, so a TPM could be used to enhance the overall system security. In public cloud computing systems, the cloud subscriber does not have direct access the TPM, so it would not be a useful resource.

Both encryption and the TPM can contribute to security of data in cloud computing, but neither offers an effective means of controlling what a user is allowed to do with data for which he or she has been granted legitimate access. Homomorphic encryption could provide some benefit, but its usefulness is limited.

Chapter 4 – Related Research

As mentioned earlier, previous UM related research, along the likes of Park and Jamkhedkar, has not yielded an enforcement mechanism that can be used in a VM. However, there is a considerable amount of related research that provides the foundation for this effort. This review first examines research on protecting information in cloud computing systems, with the objective of identifying tactics that can be applied to the UM enforcement problem. Then, previous UM research is reviewed to illustrate the role of the enforcement mechanism and to further justify the need for this capability. This is followed by a review of research on dynamically tracking data flows within a system. This capability, Data Flow Tracking (*DFT*), is often used to determine how data propagates from a network source through a system; the approach is called *taint analysis*. Because of the need to track how information flows within a VM, DFT is a capability which is necessary for UM enforcement.

4.1 Information Security in Cloud Computing Systems

In cloud computing systems, data can be encrypted while in storage and while in transit. Once it is in a VM, it must usually be decrypted to be used. The information owners are concerned that it can now be accessed by others while it is in use. The research examples shown below are intended to address this concern. One irony, as shown in the last example is that the same strategies that are used to protect data can also be used to steal data.

4.1.1 CloudVisor

CloudVisor is intended to address a cloud computing subscriber's concerns that data in the subscriber's VMs may be accessed by other users' VMs that are jointly tenant on the same set of hardware (Zhang, Chen, Chen, & Zang, 2011). It also protects data in VMs by preventing access by the cloud provider's administrators and cloud management tools. It accomplishes this by inserting a security monitor underneath the commodity VMM, in a configuration that is called *nested virtualization*. When CloudVisor is booted, it elevates the commodity VMM to execute in a less privileged mode. CloudVisor only allows an authorized VM access to unencrypted data and all other access will be directed towards an encrypted version of the data. It uses the TPM to check software integrity. For private cloud applications, it can be a viable protection concept.

4.1.2 Overshadow

Overshadow protects information in virtual machines by taking advantage of the extra level of memory mapping in the VMM that is necessary to support VMs (Chen, Garfinkel, Lewis, & Subrahmanyam, 2008). Instead of using the conventional one-to-one mapping of guest physical addresses to machine physical addresses, Overshadow uses a one-to-many mapping strategy so that, depending on context, different memory views are provided. The approach is called *multi-shadowing*. *Cloaking* uses the multi-shadowing capability to access encrypted or unencrypted versions of data, depending on context. Overshadow introduces a *shim* into the address space of a cloaked application, which

cooperates with the VMM to mediate all interactions with the operating system. The VMM identifies the guest context and maps it to an appropriate shadow page table, providing access to either encrypted or unencrypted data. Because of the role that the VMM plays in identifying guest context and switching between multiple shadow page tables, the approach is best suited to private cloud computing installations.

4.1.3 SubVirt

CloudVisor is inserted underneath a commodity VMM in order to protect data in an individual VM. In a similar manner, SubVirt inserts a VMM underneath a commodity operating system in order to provide control (King & Chen, 2006). By elevating a target operating system into a VM, the VMM now can host malicious software that cannot be detected or controlled by the operating system or application software; ultimate control of a system is in the lower levels. While this research is not devoted to developing ways to introduce malicious software, the SubVirt system clearly demonstrates how an operating system and the associated application software can be controlled (or in this case, subverted) with a VMM.

4.1.4 Cloud Information Security Implementation Considerations

Virtualization is the essential capability that enables extensible IaaS cloud computing systems. In a public IaaS system, the cloud service provider has *complete* control of the hypervisor and the underlying hardware resources; a cloud user can only implement mechanisms in the layers above the provider's hypervisor.

To address information security concerns, one approach that might be considered is writing a custom operating system and associated application software for use in the VM. This would give the user the ability to control all facets of operation, but is an unrealistic approach because today, few organizations are willing to invest in the resources needed to develop custom operating systems and applications. For example, before the mid-1990's, most US military systems were based on custom designed hardware and software. Today, the military relies extensively on Commercial-Off-The-Shelf (COTS) hardware and software products for many mission critical applications, because of lower life-cycle costs. In view of this trend of increasing reliance on COTS products, we must consider approaches that can provide the desired control without the need to modify the large universe of application software that might be used. Ideally, a control capability would also require minimal, if any, changes to the operating system software.

Cloud computing information security research is primarily focused on preventing unauthorized individuals from accessing information at rest, in transit, or while in use. Current cloud computing information security research does not address control of how data are used once a user has authorized access.

4.2 Usage Management

Usage management provides information owners an assured ability to control who has access to their data *and* an ability to control what a user can do with data, once they have been legitimately granted access. This research area

has evolved from early DRM work that was done with the objective of protecting copyrighted material that is distributed electronically.

With today's highly networked computing environment, users can easily transmit very large amounts of data in a short amount of time. Users can also quickly download data to high speed and high density media, such as thumb drives and portable disk drives. While these capabilities can be very convenient when there are no concerns about ownership or sensitivity of the data, they can also be a significant problem if owners do not want their data freely disseminated. A need for automatic control of how digital data are used has been the motivation for the research efforts described herein.

4.2.1 $UCON_{ABC}$

The $UCON_{ABC}$ work introduced a conceptual framework that moves beyond traditional access control systems, which use server-side mechanisms and an access matrix to make access decisions (Park & Sandhu, 2004). This work introduced models that integrate the *Authorizations* (A), *oBligations* (B), and *Conditions* (C) that are a foundation for UM systems. A significant extension that they proposed is the notion of continuous control of resources for which access has been granted. In their work, they noted that to provide control within a client, a client-resident trusted computing base and a reference monitor are needed for enforcement. However, they did not address this need, as their focus was on the operational model.

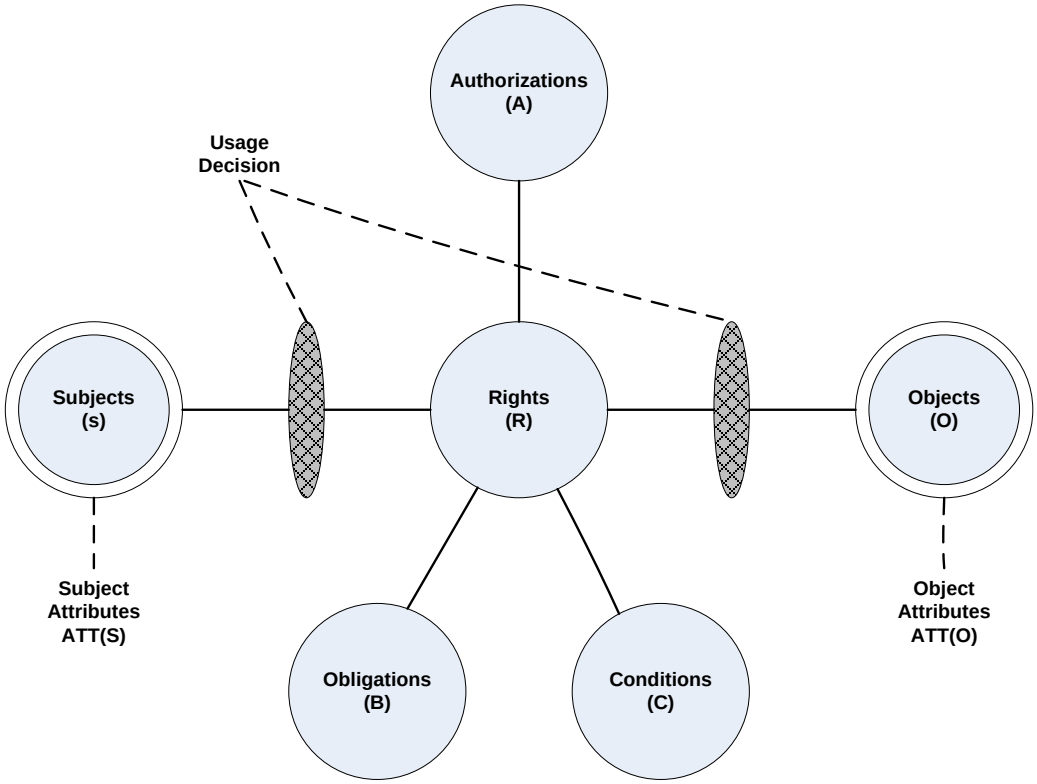

Figure 3 $UCON_{ABC}$ model components.

The $UCON_{ABC}$ work is intended to encompass the DRM capability, where the information provider retains some control over what the user can do with it. The operational models have a rich set of characteristics that can be considered in making access decisions but this discussion will focus on ongoing control. The structure is illustrated in Figure 3. The $UCON_{ABC}$ models consider the subjects, S, subject attributes, SA, objects, O, and object attributes, OA. Rights, R, are privileges that a subject can hold and exercise on an object and can include consumer rights, CR, and provider rights, PR. Authorizations, A, are functional predicates that have to be evaluated for usage decisions. oBligations, B, are functional predicates that verify mandatory requirements a subject has to

perform before or during usage. Conditions, C, are environmental or system-oriented decision factors.

We can use the $UCON_{preA0}$ model to represent the action of the data-flow-based enforcement mechanism. We use L, which is a lattice of security labels with the dominance relation, $\leq$, and functions: $: S \rightarrow L$, $maxClearance: S \rightarrow L$, and $classification: O \rightarrow L$. The lattice is first used to make the decision to allow a subject to access an object, based on the clearance level and conditions of the subject and the classification of the object. Then, the lattice can provide the information that will be used by an enforcement mechanism to govern data flows, using the following function:

$$\text{allowed}(o_1, o_2, \text{write}) \Rightarrow \text{classification}(o_1) \leq \text{classification}(o_2) \qquad (8)$$

This is consistent with the $*$ property that was presented previously.

4.2.2 An Interoperable Usage Management Framework

Jamkhedkar et al. proposed a framework for UM in open, distributed environments that emphasizes interoperability (Jamkhedkar, Heileman, & Lamb, 2010). Their system is a combination of the access control and usage control functions of the $UCON_{ABC}$ system and Digital Rights Management (*DRM*). DRM includes content management, license management, specification of usage rules, and simple access control. A key observation that they make is that the UM policies must be tightly coupled to a data resource because resources will typically be moved to locations that are not specifically known A Priori. They also recognize that each computing environment must have the capability to both interpret a policy language and enforce the policy. Their framework uses

licenses, in which the policies are expressed. The licenses are interpreted and enforced within a computation environment.

This system has two operational stages: a setup stage and a working stage. In the setup stage, the computational environment is set up and the license is generated. In the working stage, the license is interpreted as needed for enforcement in operational environment and then the policies stated in the license are enforced in the computational environment. This work was focused on the theoretical framework and did not propose any means of enforcement.

4.2.3 Usage Management in Cloud Computing

Jamkhedkar et al. later presented a concept for UM in cloud computing (Jamkhedkar, Lamb, & Heileman, 2011). This concept built on their previous design of an open, interoperable framework. They consider an operational environment consisting of systems that are operated by different cloud computing providers. The diverse set of systems necessitates a common cloud ontology so that policies can be specified and enforced consistently in each. As before, they propose a setup phase and a working phase. The setup phase uses context information from each service provider and using this information, then data set usage policies are generated. The data set usage policies are cast in the framework of the common cloud ontology. The working phase consists of policy management, interpretation, and validation. They propose a Usage Management Cloud Service that interacts with individual cloud computing systems to determine whether operations in the given contexts are permitted.

Nandina et. al. leveraged this conceptual model and implemented a hierarchical UM system for cloud computing (Nandina, et al., 2013). In this system, a centralized Usage Management Manager (*UMM*) provides the user authorization and access control decision functions. The UMM considers the user's operating context in making access control decisions. The concept of operations for the hierarchical cloud computing UM system is illustrated in *Figure 4*.

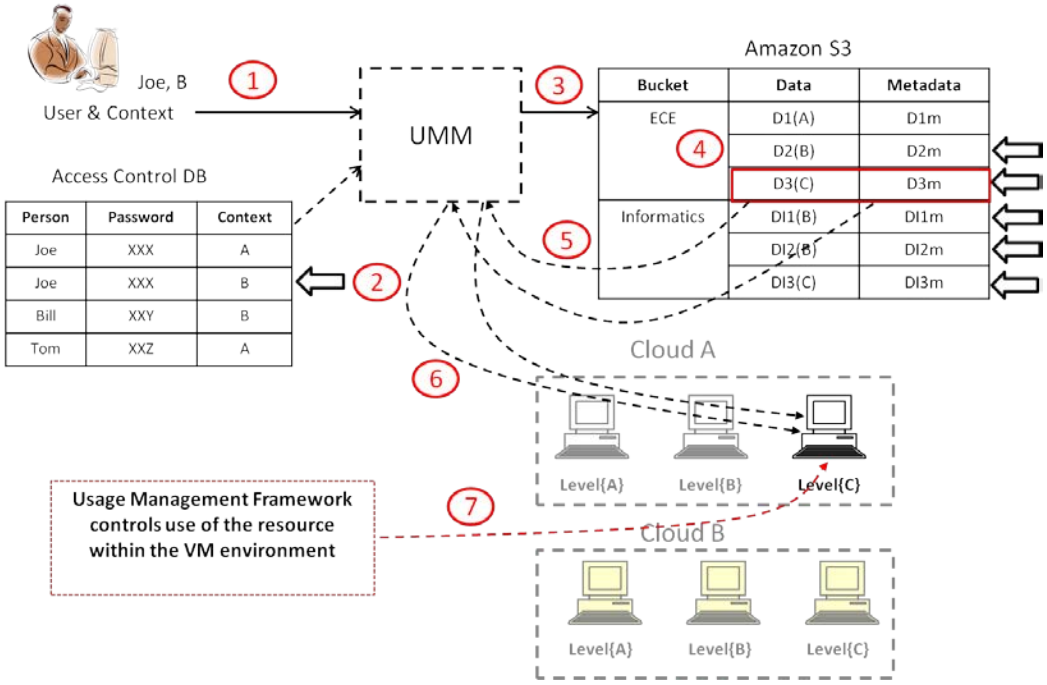

Figure 4 Usage Management in a Multi-Cloud Computing Environment

The hierarchical UM system provides the capability to control provisioning of data to VMs. The provisioning is allowed if, and only if, the requesting user is authorized to access the data in their current operating context and the target virtual machine is configured with at least the protective measures that are required for the type of information requested. The access decision implements

the simple security property as well as enforces the policy that restrict the use of sensitive information to cloud computing resources that have been configured to use particular security measures.

The hierarchical cloud computing UM system operates in the follow sequence:

1 A user connects to the UMM.

2. The UMM validates the user's credentials and desired operating context.
3. Based on data in the authorization data base, the UMM presents the user with a list of data sets that he or she is authorized to access.
4. The user selects a data set.
5. The UMM downloads both the data and a set of metadata, which we call the license. The license describes the data set characteristics and includes policy information on how the data may be used. One of the metadata fields specifies the sensitivity level of the data, which in turn corresponds to restrictions on sharing.
6. Based on the license data, the UMM instantiates a VM that is configured with protective measures that are appropriate for the data sensitivity and usage restrictions.
7. Ideally, once the data resides in the VM, a UM framework inside the VM enforces usage restrictions.

This hierarchical implementation is similar to previous work in that once data are transferred to a VM, there is no capability to actually enforce the usage

management policies within it. Certain policies, such as time of use, can be enforced by the UMM simply by shutting down the VM. Other policies, such as prohibiting any copying, cannot be enforced within the VM with this system. In order to enforce these types of policies, an enforcement mechanism must be capable of monitoring sensitive data as they enter the VM and track their movement as computation proceeds.

Given that the objective is to control the flow of information in the virtual machine, we must consider all possible flow paths. One approach to consider is using nested hypervisors. CloudVisor uses nested hypervisors. The SubVirt effort demonstrates how a hypervisor can be inserted underneath a guest operating system and can provide excellent control, with no modifications to the application software or guest operating system.

Using the hypervisor for enforcement will require controls for *every* interface to which data can be written to by the operating system. In addition, each interface may pass a mix of data, some of which must be controlled. This will require that the hypervisor-based enforcement mechanism is capable of selectively restricting data transit. To address this issue, one next considers research that is oriented toward instrumentation and monitoring information flows.

4.3 Data Flow Tracking

To determine data flows, it is possible to analyze source code and determine data flows, but that approach is logistically unfeasible. So, a means of

instrumenting applications is needed. Ideally, this must be done with no modifications to the application itself.

4.3.1 Pin

We can instrument application software using Pin and associated Pintools (Luk, et al., 2005). Pin is a software system that provides the ability to instrument application software by inserting extra code to observe its behavior. Pintools are routines which communicate with Pin and implement instrumentation and analysis functions. Pin has an extensive set of capabilities that can instrument the unmodified application at multiple levels such as: instruction, function, system call, thread, and image. It provides a capability to examine the parameters passed to functions and the corresponding returned values. One very important characteristic is that operations can be instrumented before they are actually executed, or immediately after. This is essential because if data are entering an application, the enforcement mechanism will require action immediately after the input function executes in order to properly tag incoming data. Similarly, when data are slated to exit an application, the enforcement mechanism must be invoked before the actual operation takes place to prevent prohibited actions.

4.3.2 libdft

Dynamic Data Flow Tracking (*DFT*) is generally used for *taint analysis*, which is, tracking the flow of data from a network source as it propagates in a processor. DFT research has yielded potential solutions that can be used for usage management enforcement. Specifically, libdft provides a means of

applying DFT to commodity software (Kemerlis, Portokalidis, Jee, & Keromytis, 2012). It uses Pin for instrumentation and analysis, and can provide the information necessary to selectively control how data sets are used in an application.

4.3.3 CloudFence

CloudFence is a system which uses DFT, specifically libdft, to audit the use of cloud-resident data (Pappas, Kemerlisl, Zavou, Polychronakis, & Keromytis, 2012). The system involves three parties: the cloud infrastructure provider, a cloud web service provider, and users. The intent is that CloudFence would be offered by the cloud providers to the service providers as a service; the service providers integrate the data flow tracking functions into their services and tag data that need to be protected. Then, the users can monitor the propagation of their data. The authors also suggest that service providers could potentially use the tagging information to control the flow of information after they specify the sources of sensitive data and define which paths are allowed and which are not. This would require some modification of the application software to interact with the DFT capability.

DFT is a capability that can be used for UM enforcement. As suggested by the CloudFence authors, if data of interest can be identified, then it is possible to restrict flow paths. This research effort will demonstrate that data flows can be monitored using DFT and by using licenses to identify which data are sensitive, yield an automatic UM enforcement mechanism that can be used in VMs.

Chapter 5 – Method

5.1 Test Environment

With cloud computing virtual machines, a large fraction of them use Linux for the guest operating system. An attractive characteristic of Linux (and its ancestor, Unix) is that all devices are treated as files, so moving data to and from data files and I/O devices is done using a limited set of system calls. Both software development and experiments were done on an Ubuntu 14.04 LTS 32-bit operating system executing on an Intel T4200 Dual Core Pentium processor. Instrumentation was done using the 2.13-62141-gcc.4.4.7-linux version of the Intel PIN instrumentation code. For the Pintool, the research began with the libdft-3.1415alpha Dynamic Flow Tracking software that was originally developed for taint analysis and later modified for UM enforcement. All code was compiled with gcc (Ubuntu 4.8.2-19ubuntu1) 4.8.2.

The PIN and modified libdft tools provide an operational configuration similar to what is shown in Figure 5. An unmodified and unwitting application is processed by the PIN engine and is then instrumented, analyzed, and potentially controlled using functions in a Pintool. (*libdft is a Pintool*.) In the experiments, the Pintool monitors individual instruction execution, system calls, file input and output operations, and network related functions. This provides an ability to monitor and control all input and output data flow operations of the unmodified application. This capability will ultimately be used to enforce the usage management policies within the processor.

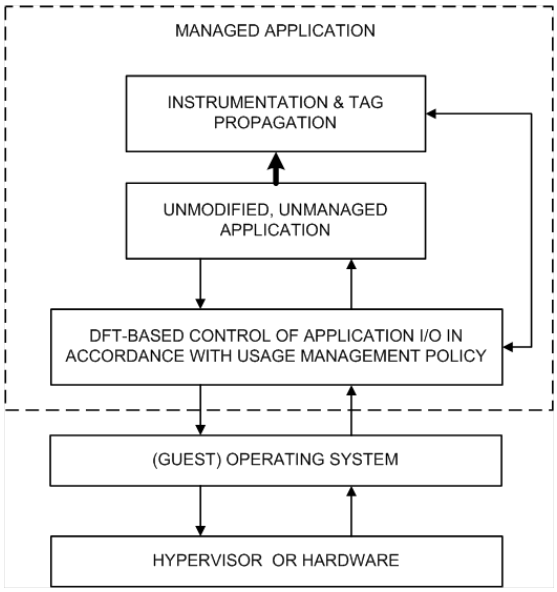

Figure 5 Enforcement Mechanism Configuration

5.2 Instrumentation

The Pin library is quite extensive in term of instrumentation capabilities. It also includes the capability to log data and events for post execution analysis. In order to verify proper operation of the libdft code after making a number of modifications to: address compatibility issues, add additional tag propagation capabilities, and extend the functionality for UM enforcement, extensive logging capabilities were added to record tag propagation operations. The log files generated during application software execution provide a way to examine, in detail, the status of all operand tags as each assembly language instruction is executed. An example of this instrumentation capability is shown in Figure 6. In this example, lines 1-6 show a sequence of assembly language instructions that are being executed. The instructions have been extracted using the Pin INS_Disassemble() instrumentation function. Next, lines 7-12 illustrate the corresponding tag propagation actions that are being performed by libdft using

Pin instruction instrumentation functions. Using a search function in an editor such as gedit, it is easy to quickly find operations of interest in the log files and analyze the tag propagation behavior.

```
 1  -- 8048698  mov edx, dword ptr [esp+0x20]
 2  -- 804869c  mov eax, dword ptr [esp+0x1c]
 3  -- 80486a0  mov dword ptr [esp+0x8], edx
 4  -- 80486a4  mov dword ptr [esp+0x4], eax
 5  -- 80486a8  mov dword ptr [esp], 0x80487db
 6  -- 80486af  call 0x8048430
 7  m2r_xfer_opl: dest reg: 5  mem src: bfb85740  dst_tag: 15  src tag: 15
 8  m2r_xfer_opl: dest reg: 7  mem src: bfb8573c  dst_tag: 15  src tag: 4080
 9  r2m_xfer_opl: mem dest: bfb85728  reg src: 5  dst_tag: 255  src_tag: 15
10  r2m_xfer_opl: mem dest: bfb85724  reg src: 7  dst_tag: 65520  src_tag: 15
11  tagmap_clrl - Mem Addr: bfb85720 memtagpost: 65520 memtagpre: 65520
12  tagmap_clrl - Mem Addr: bfb8571c memtagpost: 61440 memtagpre: 61440
```

Figure 6 Detailed Tag Propagation Instrumentation

Depending on the size of the application size and the duration of operation, the log files can be quite large. To make the log file size manageable, the detailed logging capability can be disabled, as needed.

5.3 Test Configuration

UM enforcement requires the ability to monitor data entering an application from multiple sources and controlling which, if any, devices may receive data output from the application. To test the ability of the enhanced Pintool, which is called *libdft_um,* a test application which inputs data from files, the keyboard (*stdin*), and a network interface was used. As shown in Figure 7, data entering on each interface may be tagged. As data are processed in the instrumented application, tags are propagated, as appropriate, during the

execution of each machine instruction. Eventually, the application executes an output operation and then the libdft_um instrumentation checks the data for tags and if necessary, prevents the output operation. For the experiments, alerts were recorded in the log files and the writes of tagged data were allowed to proceed.

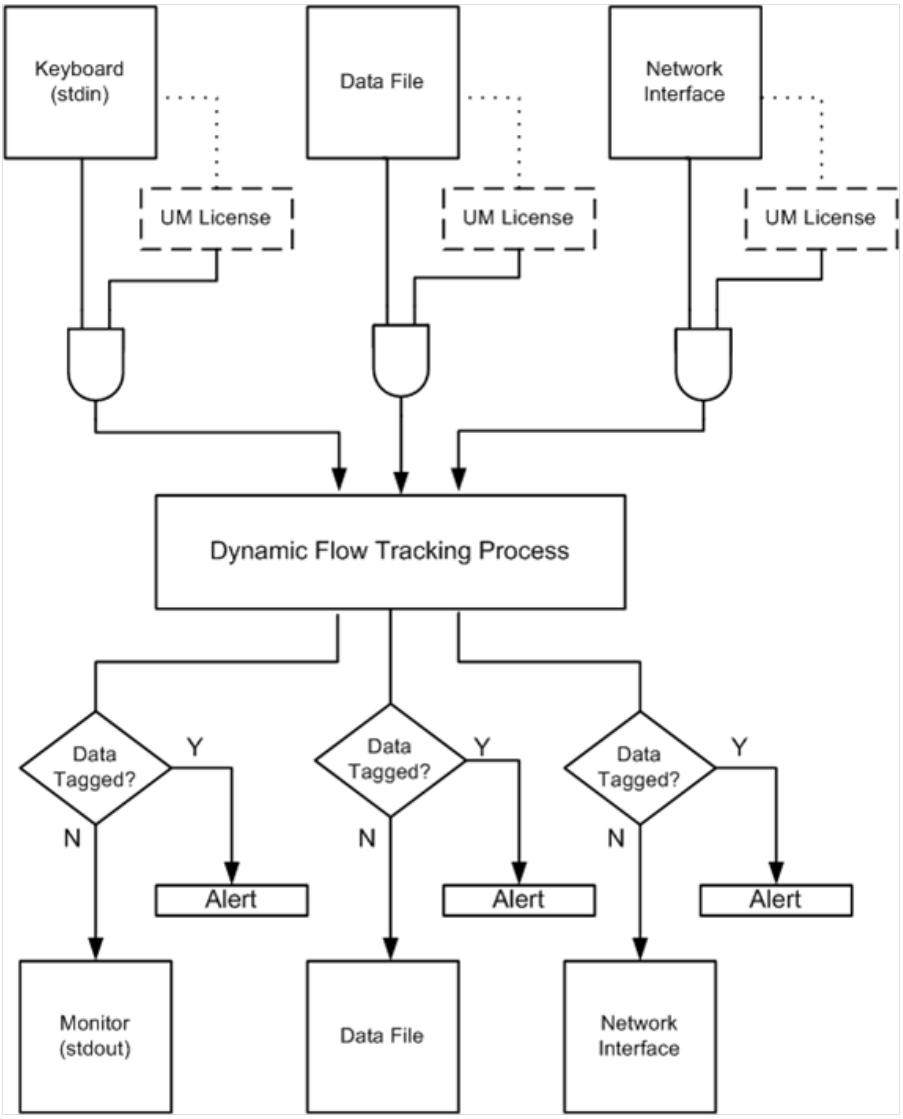

Figure 7 DFT-Based Control

5.4 Instrumentation Methodology and License Usage

The PIN system is capable of instrumenting functions either prior to execution, or after. To implement a UM enforcement mechanism, we must use both. For example, assuming that an application will read data from a file, we can first verify that the data to be read has a valid license that specifies how the information can be used. Similarly, if data are to be written to a file, we can first check that the output file has a license that specifies its sensitivity parameters. If an *open*() system call is instrumented immediately after execution, then the assigned file descriptor is available. The enforcement mechanism can then check for a corresponding license file. If the usage policy within the license designates the information as sensitive, then the file descriptor can be added to the set of "interesting inputs". If the data file has no license, then in the spirit of "*fail-safe*", the data read can automatically be treated as high sensitivity. Similarly, if an output file does not have an associated license, the output file is assumed to be a low sensitivity level. With these default settings and the dominance relationship enforcement, data will not be permitted to flow from a source with no license to an output with no license. The enforcement mechanism activities are illustrated in Figure 8 for a file which is to be read and Figure 9 for a file to which data are to be written into.

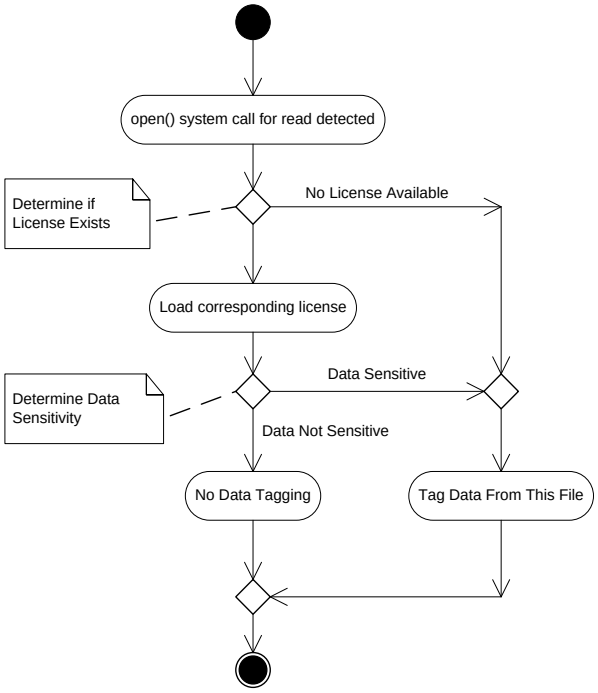

Figure 8 License check activities when *open()* is called for read operations

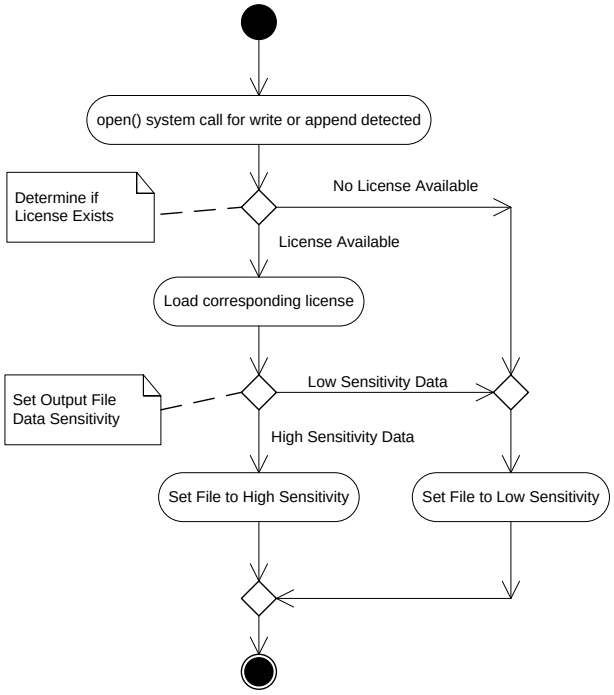

Figure 9 License check activities when *open()* is called for write operations

To enforce the data flow policies when the source and destination file licenses are provided, the enforcement mechanism checks the dominance relationship between the data and the destination and, if necessary, the enforcement mechanism can prevent the operation from proceeding. Continuing with the example in more detail, assume that the application is reading data from a file which is designated as high sensitivity. With the *read()* system call, instrumenting the function after it has been executed is more useful because the return value indicates how many bytes have actually been read, enabling the instrumentation to tag the appropriate number of tag bits. When a *write()* operation is to be executed, the instrumentation must check if the data to be written are tagged *before* the write function is actually executed. If the data are tagged, then the instrumentation function can prevent the write from proceeding. This sequence of operations is shown below in Figure 10.

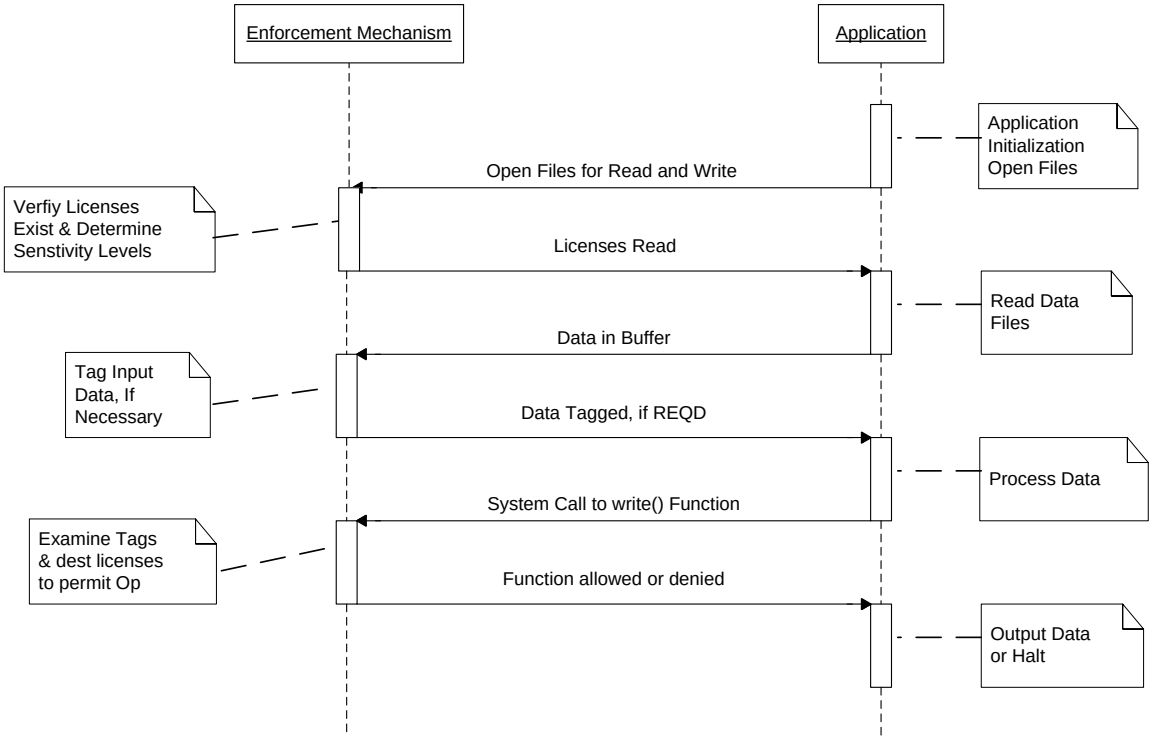

Figure 10 Enforcement Sequence

While Figure 10 refers to data files, similar operations can take place when the sources or destinations for data flows are network sockets. The individual connections must either have licenses that specify the associated sensitivity levels, or as in case of the data files, default sensitivity levels will be assumed. Again, a restrictive strategy will be used to prevent undesired data flows.

5.5 License Implementation

The enforcement mechanism must operate automatically to be useful. This requires that the enforcement mechanism must be capable of not only locating a license file, but it must also have the capability to open the file and

parse the contents to examine a specific term which specifies the data sensitivity. A typical license may use an eXtensible Markup Language (XML) variant to specify policy items. An example policy license is shown below in Figure 11 .

```
<?xml version="1.0" encoding="UTF-8"?>
  <usage_policy>
     <sensitivity_level>sensitive</sensitivity_level>
     <usage_time>days_only</usage_time>
     <usage_network>no_wireless</usage_network>
     <usage_device>no_mobile_device</usage_device>
  </usage_policy>
```

Figure 11 Typical XML-Based Policy License

In order to provide the automatic operation capability, the Libxml2 XML C parser and toolkit developed for the Gnome project were incorporated into the Pintool (Veilard, 2014). For data files, corresponding licenses were used to designate the data as sensitive, or non-sensitive. As shown in the example, the license files also contain other usage policy information that could be used by a centralized UMM to control data access by the VM.

5.6 Assessing Performance

The UM enforcement mechanism will increase the execution time of any application. The Pin instrumentation and DFT functionality must first be initialized, which includes processing the application code itself. Then as the application code is executing, the instrumentation monitors all operations including assembly language instructions and system calls. For assembly language instructions that process data, additional code for tag propagation is

also executed. For system calls associated with data input and output, license processing, tag setting, and tag condition monitoring are also performed. Because of the extensive operations performed by Pin and libdft-um, an increase in execution time is to be expected.

To get a clearer sense of the additional computational burden, testing was done with an application that reads the contents of an image file, in JPEG format, and writes it to another file. License files for both the input and output files were included, with the input designated as sensitive and the output designated as non-sensitive, to verify that the tag propagation and detection logic was functioning properly. The test application was run using image files of varying sizes and the execution time was measured while running in both instrumented and non-instrumented modes. The tests were executed in a typical Linux environment, where the operating system is executing multiple processes. However, no other user applications were executing during these tests. The tests were repeated several times, and because of multi-tasking, the execution times show some variability.

Chapter 6 - Results and Discussion

The complexity of the Pin and Pintool software led to the need for a capability to be able to examine execution data in detail after conducting an experiment. Detailed log files provide a way to verify that the tag-related operations are functioning properly and a way to record experiment results. A log file provides a time history of execution, which can then be traversed in a forward or backward direction, making it more effective than a run-time debugger. After conducting the initial experiments to verify that the data tags were being set and propagated correctly, subsequent experiments were conducted using a minimized logging mode.

The capability of using a mix of sensitive and non-sensitive data sources, was demonstrated using a test program in which the data input by the user via stdin (the keyboard) were non-sensitive and data read from both a data file and a network socket were considered sensitive. The stdout display, the output data file, and the network socket connections were all designated as non-sensitive, so if any sensitive data was to be written to these devices, the logging function would generate an alert. Information generated by the application as output to stdout for user prompts was considered as non-sensitive, as well.

Appendix A provides a set of data that was generated using this test program, which demonstrates the implemented UM enforcement mechanism properly tracking sensitive data interleaved with non-sensitive data. The UM enforcement mechanism was able to detect when sensitive data was being output and discriminate between sensitive and non-sensitive data. To

emphasize, for these tagging validation experiments, the sensitive sources were selected prior to application execution.

These tagging validation experiments included user inputs and outputs (stdin and stdout), files, and network interfaces (sockets). Similar capabilities will be needed to enforce flows in inter-process communication channels, such as pipes and shared memory; they are not implemented in this research effort.

The next set of experiments used xml license files to designate usage policies and sensitivity levels of the data files. After augmenting the UM enforcement mechanism software to include the license interpretation capabilities, subsequent testing was done using the same test program as before. Now each data file can have an associated license file of the form shown in Figure 11. As shown in Appendix B, the user dialog for this test program is identical to that of the previous set of experiments, but now the instrumentation software that implements the UM enforcement mechanism does more than just tag data and track the tag propagation.

As part of the UM enforcement process, *after* a data file is opened and assigned a file designator by the operating system, the instrumentation software opens an associated license file that is stored in same location. The instrumentation software parses the license and if the data are designated as sensitive, then the data will be tagged *after* any read from the file. If no license is available, a default designation of *sensitive* is applied to the file.

The license validation tests are significant because they demonstrate that an enforcement mechanism can automatically be selectively activated by using a license. Also very significant is the fact that the enforcement mechanism can control an application's ability to output data with no modification necessary. Because the *write()* system calls are instrumented before the system call are actually executed, the instrumentation software can prevent a write operation from actually taking place. To prevent a write operation, the Pin instrumentation can insert instructions to force an immediate return from the system call, allowing the application software to continue executing. Because the intent of this research effort is to demonstrate feasibility, the current enforcement mechanism response to writes is limited to logging prohibited actions.

The implementation strategies are illustrated in the code segments shown in Appendix C. These segments illustrate the high level instrumentation details that provide the automatic enforcement capability obtained by combining the DFT capability with machine readable licenses. The enforcement capability is possible because of the ability to instrument system calls either prior to or just after execution.

6.1 Performance Impact

With the tag propagation and license processing operation verified, the next question to address was: "what is the actual processing burden imposed by the enforcement mechanism?" To minimize the effects of communication delays and user interaction, the UM enforcement mechanism was tested using an application that copies data from one file source and writes it to another file

destination. With this test application, none of the file data are copied to stdout. Multiple experiments were conducted using various size files. The timing results are shown below in Table 1 .

	Input File Size	Non-Instrumented Execution time nS	Instrumented Execution time nS
1	9.2kB	499088	519161522
2	9.2kB	339202	516847624
1	10 x 9.2kB	2615393	549511587
2	10 x 9.2kB	2580500	562787186
1	2.6MB	14270359	545390960
2	2.6MB	13440185	561819342
1	10 x 2.6MB	118059117	701829666
2	10 x 2.6MB	118275120	680895333

Table 1 - Measured Execution Times

Each timing experiment was conducted twice to illustrate the variation that is experienced because of execution scheduling by the operating system. This is typical of any multi-process environment. The next thing to note is that there is a fixed amount of time that is used to initialize the Pin instrumentation, as it pre-processes the application executable to provide the run-time monitoring capability. The Pintool, or libdft-um, must also be initialized, which adds to the initialization time.

As seen below in Figure 12, the execution time grows linearly, as a function of the quantity of data transferred, in both the instrument and non-instrumented application tests. This test program did not have any data input from stdin (the user) or from a network interface nor was there was any output to stdout or to a network interface. As a result, there were no input or output delays that would increase the execution time, so the test program does give reasonable insight into the delays introduced by the instrumentation programs used to implement the enforcement mechanism.

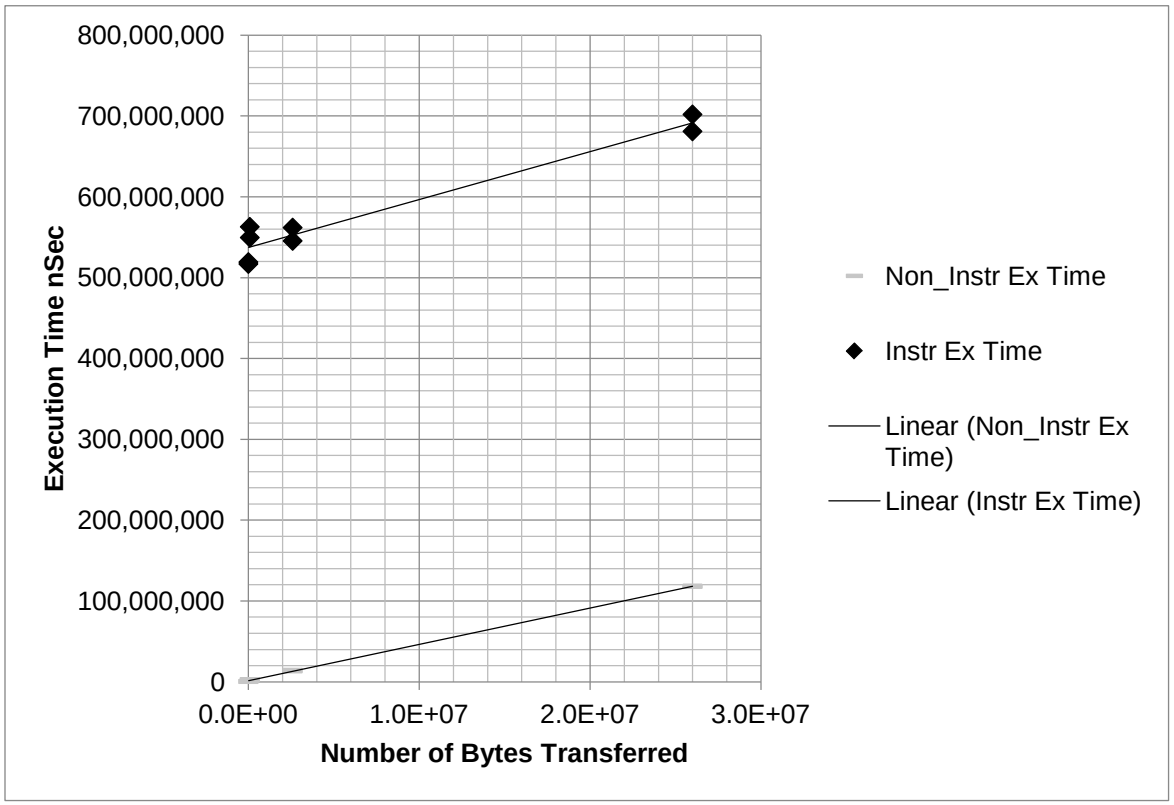

Figure 12 - Timing Comparison Test Results

In evaluating the timing test results, it must be emphasized that a UM enforcement mechanism would *only* be used in VMs that are intended to process

either a mix of sensitive and non-sensitive data. Again, the purpose is to restrict what a user can do with data once he or she has been granted legitimate access. In this situation, the performance penalty associated with initialization would be acceptable.

Keeping in mind that the UM enforcement mechanism is only intended for use in limited situations, one might consider an implementation of where the information owner instantiates a VM with an appropriately configured image of the guest operating system and application software. (This is an example of the IaaS service model.) The operator could then start execution of UM enforced applications, with which the users would then interact. From the user perspective, this would be the Software as a Service (SaaS) model of operation, which is typical of users' everyday interactions with commercial cloud-based services.

6.2 Thoroughness of Tag Testing

Output functions often aggregate data incrementally in an output buffer, before an actual write to the device takes place. To deal with a situation of where the stream of data into a buffer has sensitive data interleaved with non-sensitive data, the enforcement mechanism examines the entire contents of the buffer to verify that all the data is non-sensitive before the actual write takes place. This approach prevents inserting sensitive data in the middle of a stream of non-sensitive data.

6.3 Implicit Data Flows

A last issue to address is that of implicit data flows. In many applications, data are read in, modified using arithmetic or logic operations, and then modified data are written out. This type of flow is common, but one cannot assume that only explicit data flows are of interest when implementing a UM enforcement capability.

In the early stages of trying to apply the DFT software for UM enforcement, a number of experiments resulted in situations where data tags did not propagate as expected. This provided some of the motivation to add the comprehensive logging capability for detailed analysis. The analysis brought to light the need for some software changes to extend the tag propagation capabilities for UM. Yet, there were still cases where tags were not propagating as expected. Further detailed analysis of the assembly language instruction instrumentation log data revealed that in the application, some library data format conversion routines use conditional branching, based on input data values, to generate equivalent values. The specific case where this implicit flow was found was in the conversion process between integer values to an equivalent ASCII string that was to be written to a data file. This discovery leads one to conclude that a strategy of propagating tags at the assembly language instruction level only, may not be adequate to track implicit data flows.

As stated earlier, one of the goals of this research is to be able to control how data are used by application software without the need for modifications to the source code. Application source code can be analyzed to identify and track

all data flows, but this may not be feasible, nor desirable. Also, almost all applications use standard library functions for many purposes. So even if the application source code does not contain implicit data flows in the design, there is a possibility that the library functions do contain such flows, as this research experience has shown.

The dynamic instrumentation capability obtained by using Pin provides a means of monitoring all instructions of the application source code and associated library functions. Enhancing the Pin-based UM enforcement mechanism to detect implicit data flows will provide a more comprehensive capability.

To detect the implicit data flows, tag propagation might be done at a higher level, such as at the function level. (Fenton, 1974) described an abstract Data Mark Machine to study implicit data flows and proposed tagging the program counter to convert implicit data flows to explicit data flows. The theoretical approach was based on a modified Minsky machine. In the approach, two objectives are to ensure that a non-sensitive execution path is not dependent on sensitive information and when operating in a sensitive execution path, to ensure that no non-sensitive registers can be changed. While the specific analysis approach is not directly applicable, insights gleaned from this work may guide an implementation strategy.

Conceptually, one could consider enhancing the UM enforcement mechanism in a similar manner. This could be done by adding tag propagation logic where, if any data passed into a routine is sensitive, then output data is also

considered sensitive and tagged accordingly. This logic could be invoked when detecting a *call* instruction. If any of the parameters passed to the routine are tagged, then when the *ret* instruction is detected, registers used to return parameters or memory pointers would also be tagged. Suffice it to say; adding an ability to track implicit flows will make this demonstrated UM enforcement mechanism more effective.

Conclusions

The results of this research show that while using a completely software-based approach, it is possible to automatically enforce usage management within a processor environment. The enforcement mechanism can use data flow tracking to monitor data flows within an unmodified application and identify and prevent unwanted flows to a variety of destinations. Thus, the enforcement mechanism can control which actions the user is allowed to execute for a specific set of data. Because the approach does not require direct access to any hardware, it can be used effectively in a virtual machine, and by extension, in a cloud computing environment that provides resources using the Infrastructure as a Service operating model.

Automatic operation of the enforcement mechanism requires machine readable licenses for every data set to be controlled. The licenses must specify usage policies and in the absence of a license, there must be a default policy specified. This research effort has demonstrated an enforcement mechanism capable of using license-specified policies to identify information that must be controlled.

The enforcement mechanism does impose a performance penalty. Experimental results show that there are two components, one a fixed delay associated with the initialization process, and a second component that grows linearly with the size of the data sets to be processed, similar to the unmodified application. Thus, an enforcement mechanism should not present an undue processing burden.

Ideally, this usage management enforcement mechanism capability can be used with *any* unmodified application software. There should be no need to analyze the application software and the associated libraries, as all explicit data flows will be detected automatically. However, there is no way to ensure that the software developers do not use data conversion algorithms that contain implicit data flows. Therefore, for greater assurance that the enforcement mechanism does not allow users to use data in a prohibited manner, this enforcement mechanism capability should be extended to detect implicit data flows.

Appendix A - Tagging Validation Experiments

This appendix contains test results of experimentation that was done to verify the proper operation of the tag setting, propagation, and detection capabilities of the UM enforcement mechanism. In this experiment, the system was configured to read from: a data file, a network socket operating as a server, and the keyboard (stdin). A second process was executing a client program that provided the input data for the server. After outputting a set of user prompts, the application program read one line of data from the specified input source. Only the data input from the network socket and data file were tagged. The dialog between the application and the user via stdin and stdout (file descriptors 0 and 1, respectively) is shown below. Note: For brevity, The dialog associated with the user prompts is omitted after the first instance.

```
ejnava@ejnava-HP-G60-Notebook-PC:~/libdft/libdft_linux-i386/tools$ sudo
/usr/src/pin/pin -follow-execv -t libdft-um.so -s 0 -- ./file_io5c

------------------------------------
Select which input source to use:
1 - keyboard
2 - data file
3 - network
4 - Quit
-1
input data:
here is an input from the keyboard
here is an input from the keyboard
------------------------------------
-2
this is a file with test data used for experimenting with system calls.
------------------------------------
-1
input data:
here is another input from the keyboard
here is another input from the keyboard
------------------------------------
-3
waiting for network data
```

```
Received packet from 127.0.0.1:52330
Data: Here is a message from a client to the server
-----------------------------------
-3
waiting for network data
Received packet from 127.0.0.1:52330
Data: Here is another message from the client to the server
-----------------------------------
-2
It contains multiple lines that will be read in an interleaved fashion
-----------------------------------
-4
```

All data that are input from: stdin, the data file, and the network interface, are written to an output file. For this experiment, no writes are prohibited, but the tag status of all data written to any interface is logged in the log file. When data are input on stdin, they are echoed back to stdout. When a line of data is read from a data file, the line is written to stdout and to the output file. When a line of data is received on the network interface, the data are written to stdout, to the output file, and are also echoed back to the client through the network interface.

Shown below are excerpts from the log file generated during execution of the experimental application code with the full logging capability enabled. Not shown are instruction disassembly and tag propagation logging. Also not shown are the repetitive instances of log entries associated with the output of the options menu and input prompt sent to stdout for each input sequence. The log operations are triggered by instrumentation when the application is making a system call. Each system call type has a unique integer identifier. For the system calls shown in the log file below, the integer identifiers are given in Table 2.

Syscall Number	Description
3	sys_read
4	sys_write
5	sys_open
6	sys_close
33	sys_access
45	sys_brk
91	sys_munmap
102	sys_socketcall
125	sys_mprotect
192	sys_mmap_pgoff
197	sys_fstat64
243	sys_set_thread_area
252	sys_exit_group

Table 2 Select System Call Numbers

For the sys_socketcall, a specific function requested is specified using an integer function number as part of the system call. As one system call is used for all socket functions, it necessary to also record the function number for analysis. The Socketcall function numbers used in the logfile excerpt below are shown in Table 3.

Socketcall Function Number	Description
1	sys_socket
2	sys_bind
11	sys_sendto
12	sys_recvfrom

Table 3 Select Socketcall Function Numbers

The excerpts from the logfile are shown below. The number on the left is the line number of the original logfile. On each line is information that has been logged by the enforcement mechanism software. Note that what is shown has been

generated with the full instrumentation capability enabled. For normal operation, the logging is not as extensive.

```
--- Excerpts from pintool.log of Experiment on Oct 23,2014

 63834++ syscall: 102
 63838-------- pre_socketcall_hook - function no: 1
 63836-------- post_socketcall_hook - function no: 1
 63837       Adding socket descriptor fn: 4 to monitored set

 65637++ syscall: 102
 65638-------- pre_socketcall_hook - function no: 2
 65639-------- post_socketcall_hook - function no: 2

 107538      ++ syscall: 5
 107539      ..... post_open_hook - fn: 5

 108154       ++ syscall: 5
 108155       ..... post_open_hook - fn: 6

 109710      ++ syscall: 4
 109711      .........entering pre_write_hook fn: 1
 109712      ---> pre_write_hook: tagmap_issetn test - Address: 95988000
Size: 1
 109713      .........entering post_write_hook fn: 1
 109714      Entering tagmap_clrn - address:95988000 size:2

 --------- Start of User Prompts Logging -----------

 111338      ++ syscall: 4
 111339      .........entering pre_write_hook fn: 1
 111340      ---> pre_write_hook: tagmap_issetn test - Address: 95988000
Size: 37
 111341      .........entering post_write_hook fn: 1
 111342      Entering tagmap_clrn - address:95988000 size:38

 111703      ++ syscall: 4
 111704      .........entering pre_write_hook fn: 1
 111705      ---> pre_write_hook: tagmap_issetn test - Address: 95988000
Size: 35
 111706      .........entering post_write_hook fn: 1
 111707      Entering tagmap_clrn - address:95988000 size:36

 111991      ++ syscall: 4
 111992      .........entering pre_write_hook fn: 1
 111993      ---> pre_write_hook: tagmap_issetn test - Address: 95988000
Size: 14
 111994      .........entering post_write_hook fn: 1
 111995      Entering tagmap_clrn - address:95988000 size:15

 112263      ++ syscall: 4
```

```
 112264      .........entering pre_write_hook fn: 1
 112265      ---> pre_write_hook: tagmap_issetn test - Address: 95988000
Size: 15
 112266      .........entering post_write_hook fn: 1
 112267      Entering tagmap_clrn - address:95988000 size:16

 112509      ++ syscall: 4
 112510      .........entering pre_write_hook fn: 1
 112511      ---> pre_write_hook: tagmap_issetn test - Address: 95988000
Size: 13
 112512      .........entering post_write_hook fn: 1
 112513      Entering tagmap_clrn - address:95988000 size:14

 113437      ++ syscall: 4
 113438      .........entering pre_write_hook fn: 1
 113439      ---> pre_write_hook: tagmap_issetn test - Address: 95988000
Size: 10
 113440      .........entering post_write_hook fn: 1
 113441      Entering tagmap_clrn - address:95988000 size:11

 114459      ++ syscall: 4
 114460      .........entering pre_write_hook fn: 1
 114461      ---> pre_write_hook: tagmap_issetn test - Address: 95988000
Size: 1
 114462      .........entering post_write_hook fn: 1
 114463      Entering tagmap_clrn - address:95988000 size:2

 --------- End of User Prompts Logging -----------

 114618      ++ syscall: 3
 114619      ......post_read_hook fn: 0
 114620             post_read_hook: not an interesting source - clear tags
 114621      Entering tagmap_clrn - address:95967000 size:2

 114959      ++ syscall: 4
 114960      .........entering pre_write_hook fn: 1
 114961      ---> pre_write_hook: tagmap_issetn test - Address: 95988000
Size: 13
 114962      .........entering post_write_hook fn: 1
 114963      Entering tagmap_clrn - address:95988000 size:14

 115623      ++ syscall: 3
 115624      ......post_read_hook fn: 0
 115625             post_read_hook: not an interesting source - clear tags
 115626      Entering tagmap_clrn - address:95967000 size:35

 118180 ++ syscall: 4
 118181 .........entering pre_write_hook fn: 1
 118182 ---> pre_write_hook: tagmap_issetn test - Address: 95988000 Size:
35
 118183 .........entering post_write_hook fn: 1
 118184 Entering tagmap_clrn - address:95988000 size:36
```

```
....  User prompts logging

121262`++ syscall: 3
121263     ......post_read_hook fn: 0
121264            post_read_hook: not an interesting source - clear tags
121265     Entering tagmap_clrn - address:95967000 size:2

122067     ++ syscall: 3
122068     ......post_read_hook fn: 5
122069     .....post_read_hook: tagmap_setn addr: 95942000 size: 402
122070     ........Entering tagmap_setn - address:95942000 size:402
122071      ***** Tagged data verified

122616`++ syscall: 4
122617     .........entering pre_write_hook fn: 1
122618     ---> pre_write_hook: tagmap_issetn test - Address: 95988000
Size: 72
122619     ***** Tagged data being written at Address 95988000 Size: 72
122620     ********* ALERT!! ***********
122621     .........entering post_write_hook fn: 1
122622     Entering tagmap_clrn - address:95988000 size:73

123142     ++ syscall: 4
123143     .........entering pre_write_hook fn: 1
123144     ---> pre_write_hook: tagmap_issetn test - Address: 95988000
Size: 1
123145     .........entering post_write_hook fn: 1
123146     Entering tagmap_clrn - address:95988000 size:2

126224     ++ syscall: 3
126225     ......post_read_hook fn: 0
126226            post_read_hook: not an interesting source - clear tags
126227     Entering tagmap_clrn - address:95967000 size:2

126481     ++ syscall: 4
126482     .........entering pre_write_hook fn: 1
126483     ---> pre_write_hook: tagmap_issetn test - Address: 95988000
Size: 13
126484     .........entering post_write_hook fn: 1
126485     Entering tagmap_clrn - address:95988000 size:14

126683     ++ syscall: 3
126684     ......post_read_hook fn: 0
126685            post_read_hook: not an interesting source - clear tags
126686     Entering tagmap_clrn - address:95967000 size:40

127542     ++ syscall: 4
127543     .........entering pre_write_hook fn: 1
127544     ---> pre_write_hook: tagmap_issetn test - Address: 95988000
Size: 40
127545     .........entering post_write_hook fn: 1
```

```
 127546     Entering tagmap_clrn - address:95988000 size:41

 ....  User prompts logging

 130624     ++ syscall: 3
 130625     ......post_read_hook fn: 0
 130626            post_read_hook: not an interesting source - clear tags
 130627     Entering tagmap_clrn - address:95967000 size:2

 130891     ++ syscall: 4
 130892     .........entering pre_write_hook fn: 1
 130893     ---> pre_write_hook: tagmap_issetn test - Address: 95988000
Size: 25
 130894     .........entering post_write_hook fn: 1
 130895     Entering tagmap_clrn - address:95988000 size:26

 131960     ++ syscall: 102
 131961     -------- pre_socketcall_hook - function no: 12
 131962     -------- post_socketcall_hook - function no: 12
 131963            SYS_RECVFROM - tagmap_setn
 131964     ........Entering tagmap_setn - address:bf94b68c size:46

 135843     ++ syscall: 4
 135844     .........entering pre_write_hook fn: 1
 135845     ---> pre_write_hook: tagmap_issetn test - Address: 95988000
Size: 37
 135846     .........entering post_write_hook fn: 1
 135847     Entering tagmap_clrn - address:95988000 size:38

 136251     ++ syscall: 4
 136252     .........entering pre_write_hook fn: 1
 136253     ---> pre_write_hook: tagmap_issetn test - Address: 95988000
Size: 52
 136254     ***** Tagged data being written at Address 95988000 Size: 52
 136255     ********* ALERT!! ***********
 136256     .........entering post_write_hook fn: 1
 136257     Entering tagmap_clrn - address:95988000 size:53

 136830     ++ syscall: 102
 136831     -------- pre_socketcall_hook - function no: 11
 136832           SYS_SENDTO - Buffer: bf94b68c size: 46
 136833           ***** Tagged data being written at Address bf94b68c
Size: 46
 136834           ********* ALERT!! ***********
 136835     -------- post_socketcall_hook - function no: 11

 137216     ++ syscall: 4
 137217     .........entering pre_write_hook fn: 1
 137218     ---> pre_write_hook: tagmap_issetn test - Address: 95988000
Size: 1
 137219     .........entering post_write_hook fn: 1
 137220     Entering tagmap_clrn - address:95988000 size:2
```

```
....  User prompts logging

 140298     ++ syscall: 3
 140299     ......post_read_hook fn: 0
 140300            post_read_hook: not an interesting source - clear tags
 140301     Entering tagmap_clrn - address:95967000 size:2

 140563     ++ syscall: 4
 140564     .........entering pre_write_hook fn: 1
 140565     ---> pre_write_hook: tagmap_issetn test - Address: 95988000
Size: 25
 140566     .........entering post_write_hook fn: 1
 140567     Entering tagmap_clrn - address:95988000 size:26

 140685     ++ syscall: 102
 140686     -------- pre_socketcall_hook - function no: 12
 140687     -------- post_socketcall_hook - function no: 12
 140688            SYS_RECVFROM - tagmap_setn
 140689     ........Entering tagmap_setn - address:bf94b68c size:54

 142586     ++ syscall: 4
 142587     .........entering pre_write_hook fn: 1
 142588     ---> pre_write_hook: tagmap_issetn test - Address: 95988000
Size: 37
 142589     .........entering post_write_hook fn: 1
 142590     Entering tagmap_clrn - address:95988000 size:38

 142990     ++ syscall: 4
 142991     .........entering pre_write_hook fn: 1
 142992     ---> pre_write_hook: tagmap_issetn test - Address: 95988000
Size: 60
 142993     ***** Tagged data being written at Address 95988000 Size: 60
 142994     ********* ALERT!! ***********
 142995     .........entering post_write_hook fn: 1
 142996     Entering tagmap_clrn - address:95988000 size:61

 143136     ++ syscall: 102
 143137     -------- pre_socketcall_hook - function no: 11
 143138           SYS_SENDTO - Buffer: bf94b68c size: 54
 143139           ***** Tagged data being written at Address bf94b68c
Size: 54
 143140           ********* ALERT!! ***********
 143141     -------- post_socketcall_hook - function no: 11

 143509     ++ syscall: 4
 143510     .........entering pre_write_hook fn: 1
 143511     ---> pre_write_hook: tagmap_issetn test - Address: 95988000
Size: 1
 143512     .........entering post_write_hook fn: 1
 143513     Entering tagmap_clrn - address:95988000 size:2
```

```
 ....  User prompts logging

 146591     ++ syscall: 3
 146592     ......post_read_hook fn: 0
 146593            post_read_hook: not an interesting source - clear tags
 146594     Entering tagmap_clrn - address:95967000 size:2

 147175     ++ syscall: 4
 147176     .........entering pre_write_hook fn: 1
 147177     ---> pre_write_hook: tagmap_issetn test - Address: 95988000
Size: 71
 147178     ***** Tagged data being written at Address 95988000 Size: 71
 147179     ********* ALERT!! ***********
 147180     .........entering post_write_hook fn: 1
 147181     Entering tagmap_clrn - address:95988000 size:72

 147680     ++ syscall: 4
 147681     .........entering pre_write_hook fn: 1
 147682     ---> pre_write_hook: tagmap_issetn test - Address: 95988000
Size: 1
 147683     .........entering post_write_hook fn: 1
 147684     Entering tagmap_clrn - address:95988000 size:2

 ....  User prompts logging

 150762     ++ syscall: 3
 150763     ......post_read_hook fn: 0
 150764            post_read_hook: not an interesting source - clear tags
 150765     Entering tagmap_clrn - address:95967000 size:2

 151257     ++ syscall: 6
 151258     --------- post_close_hook: 4

 152021     ++ syscall: 6
 152022     --------- post_close_hook: 5

 153079     ++ syscall: 4
 153080     .........entering pre_write_hook fn: 6
 153081     ---> pre_write_hook: tagmap_issetn test - Address: 95953000
Size: 318
 153082     ***** Tagged data being written at Address 95953000 Size: 318
 153083     ********* ALERT!! ***********
 153084     .........entering post_write_hook fn: 6
 153085     Entering tagmap_clrn - address:95953000 size:319

 153152     ++ syscall: 6
 153153     --------- post_close_hook: 6
```

A detailed explanation of the results follows:

- Lines 63834 – 63837 are logging the creation of the socket that is used for network communication. The socket is assigned file descriptor 4.
- Lines 65637 – 65639 are logging the action of binding a local address to the socket.
- Lines 107538 – 107539 are logging the opening of a data file to be read. This file is assigned file descriptor 5.
- Lines 108154 – 108155 are logging the opening a file to which data are to be written. It is assigned file descriptor 6.
- Lines 109710 – 109714 are logging the writing of a program generated character, Newline, to stdout (file descriptor 1, by default). The character is not tagged, as it does not come from a sensitive source.
- Lines 111338 – 111342, 111703 – 111707, 111991 – 111995, 112263 – 112267, 112509 – 112513, 113437 – 113441, and 114459 – 114463 are the logging associated with writing the options menu and input prompt to stdout. These are generated by the application and the data are not tagged. This sequence repeats and is not shown in the rest of the application dialog or log data under discussion.
- Lines 114618 – 114621 are logging the data that is input to stdin (file descriptor 0, by default). This is the input from the user which selects which data source to read. For this experiment, data input into stdin is not marked as sensitive, so it is not tagged.
- Lines 114959 – 114963 are logging the data written to stdout to prompt the user to input data after having selected option 1.

- Lines 115623 – 115626 are logging the application reading data from stdin that the user is inputting. This line of data will later be written to an output file. Because in this experiment, stdin is not considered a sensitive source, the data are not tagged.
- Lines 118180 – 118184 are logging of the application software writing of the data, which was input by the user into stdin, back out to stdout. Note that in the user dialog, there are two copies of the data input by the user because the operating system echoes back what is typed in, as well. Before the data are written to stdout, they are checked to see if any of the data are tagged. None are, so there is no alert.
- Lines 121262 – 121265 are logging the data input to stdin by the user to select the next input source. This input is not tagged.
- Lines 122067 – 122071 are logging the reading from the data file which has been designated as a sensitive source. This file, which has file descriptor 5 as its identification, is read as one large block by the operating system for efficiency. As shown, the data are tagged.
- Lines 122616 – 122622 are logging the writing of the first line of data read from fd:5 to stdout. As always, all data are checked for tags before the write is executed, and the enforcement mechanism detects the presence of tagged data and generates an alert.
- Lines 123142 – 123146 are logging the writing of the user prompt character to stdout. As before, the application generated data are not tagged.

- Lines 126624 – 126627 are logging the data input to stdin by the user to select the next input source. Again this input is not tagged.
- Lines 126481 – 126485 are logging the data written to stdout to prompt the user to input data after having selected option 1.
- Lines 126683 – 126686 are logging the application reading data from stdin. The data are not tagged.
- Lines 127542 – 127546 are logging the application writing data that were just read into stdin back out to stdout. There are two copies of the data on the application dialog because of the echo action described above. The data are not tagged because the source is stdin.
- Lines 130624 – 130627 are logging the user input into stdin that selects the next data input source. The input is not tagged.
- Lines 130891 – 130893 are logging the application writing the message “waiting for network data” to stdout. This is internally generated and is not tagged.
- Lines 131960 – 131964 are logging the input of data from the network interface. For the experiment, the network interface is configured as a source of sensitive information. All received data are tagged.
- Lines 135843 – 135847 are logging a message generated by the application that reports receipt of a network message from ip_address:port_no. The message does not include any of the input data so the data written to stdout are not tagged.

- Lines 136251 – 136257 are logging the write of data received from the network interface out to stdout. As the data are from a sensitive source, the data are checked for tags. Tagged data are detected and an alert is issued.
- Lines 136830 – 136835 are logging the writing of data, which was previously read from the network interface, back out to the network interface to the remote client. The data are checked for tags, which are detected, and an alert is issued accordingly.
- Lines 137216 – 143141 log a repeat of the sequence of a user selecting the network interface, receipt of data from the network interface, and re-transmission of that data to stdout and to the network interface with the same alert notifications as before.
- Lines 143509 – 147181 log a repeat of the sequence of a user selecting the data file for input (which is considered sensitive) and writing the data back out to stdout.
- Lines 147680 – 147684 log the application generating a blank line and user prompt to stdout. The data are not tagged.
- Lines 150762 – 150765 log the user inputting a quit command from stdin. The input is not sensitive, so data are not tagged.
- Lines 151257 – 151258 and lines 152021 – 152022 log the close operation for the network connection and for the input data file.

  It is important to note that all data that is being input to the application is being written to the output file referenced by fd:6. Up to this point no

write() system calls have been executed for this fd. The reason is that data are being copied to a buffer. Up to this point, no data in the buffer have been written to the file.

- Lines 153079 – 153085 are logging the application write of data to the output file with fd:6. The data include a mix of tagged and untagged data. Each byte in the buffer with valid data to be written is tested for tags. If any of these byes are tagged, then an alert is issued.
- Lines 153152 – 153153 are logging the closing of the output file.

As shown by these experimental results, the enforcement mechanism can accurately track explicit flows of data from sensitive sources to all outputs. The mechanism effectively discriminates between non-sensitive and sensitive data.

Appendix B – License Parsing Validation Experiments

The purpose of the next set of experiments was to demonstrate the correct parsing of license files associated with the data files, and to demonstrate that the enforcement mechanism could tag data based on the license file contents. The UM enforcement mechanism is completely independent of the application, so this set of experiments used the same multi-input source, multi-output destination test application as was used for the tag propagation validation experiments summarized in Appendix A.

The experiments were intended to demonstrate the proper use of licenses to determine the sensitivity of data contained in data files. The user dialog in the case of a test application reading a data file proceeds as follows:

```
ejnava@ejnava-HP-G60-Notebook-PC:~/libdft/libdft_linux-i386/tools$ sudo
/usr/src/pin/pin -follow-execv -t libdft-um.so -s 0 -- ./file_io5c

-------------------------------------
Select which input source to use:
1 - keyboard
2 - data file
3 - network
4 - Quit
-2
this is a file with test data used for experimenting with system calls.

-------------------------------------
Select which input source to use:
1 - keyboard
2 - data file
3 - network
4 - Quit
-4
```

In this case, the user selects a data file input source and the first line is read and output to the terminal (stdout). Next, the user terminates the application by selecting the option 4. To illustrate the details of the operations that are taking

place and are monitored by the enforcement management mechanism, the entire logfile contents are given below. (Note that the detailed logging functionality has been turned off, resulting in a considerably smaller log file.)

```
 1   Pin 2.13 kit 62139
 2    tagmap allocated - address:95dbf000 size:536870912
 3     - Invalid base and index registers
 4    ++ syscall: 45
 5    ++ syscall: 33
 6    ++ syscall: 192
 7    ++ syscall: 33
 8    ++ syscall: 5
 9    ++ syscall: 197
10    ......sysexit_save call to tagmap_clrn
11    +++++++++ sys call id: 197  addr: bf9a0eb0 size: 96
12    ++ syscall: 192
13    ++ syscall: 6
14    --------- post_close_hook: 4
15    ++ syscall: 33
16    ++ syscall: 5
17    ++ syscall: 3
18    ......post_read_hook fn: 4
19    post_read_hook: not an interesting source - clear tags
20    .....post_read_hook: tagmap_clrn addr: bf9a1000 size: 512
21    ++ syscall: 197
22    ......sysexit_save call to tagmap_clrn
23    +++++++++ sys call id: 197  addr: bf9a0f20 size: 96
24    ++ syscall: 192
25    ++ syscall: 192
26    ++ syscall: 192
27    ++ syscall: 6
28    --------- post_close_hook: 4
29    ++ syscall: 192
30    ++ syscall: 243
31    ++ syscall: 125
32    ++ syscall: 125
33    ++ syscall: 125
34    ++ syscall: 91
35    ++ syscall: 102
36    -------- pre_socketcall_hook - function no: 1
37    -------- post_socketcall_hook - function no: 1
38   Adding socket descriptor fn: 4 to monitored set
39    ++ syscall: 102
40    -------- pre_socketcall_hook - function no: 2
41    -------- post_socketcall_hook - function no: 2
42    ++ syscall: 45
43    ++ syscall: 45
44    ++ syscall: 5
45    ..... post_open_hook - fn: 5
46    ......opened filename: test_data.txt
47    .......examining license file: test_data.txt.lic
48    .......file - test_data.txt.lic - is sensitive
```

```
 ++ syscall: 5
 ..... post_open_hook - fn: 6
 ......opened filename: test_data2.txt
 .......examining license file: test_data2.txt.lic
 .......file - test_data2.txt.lic - is not sensitive
 ++ syscall: 197
 ......sysexit_save call to tagmap_clrn
 +++++++++ sys call id: 197  addr: bf9a1280 size: 96
 ++ syscall: 192
 ++ syscall: 4
 .........entering pre_write_hook fn: 1
 ---> pre_write_hook: tagmap_issetn test - Address: 95817000 Size: 1
 .........entering post_write_hook fn: 1
post_write_hook: tagmap_clrn address: 95817000 size: 1
 ++ syscall: 4
 .........entering pre_write_hook fn: 1
 ---> pre_write_hook: tagmap_issetn test - Address: 95817000 Size: 37
 .........entering post_write_hook fn: 1
post_write_hook: tagmap_clrn address: 95817000 size: 37
 ++ syscall: 4
 .........entering pre_write_hook fn: 1
 ---> pre_write_hook: tagmap_issetn test - Address: 95817000 Size: 35
 .........entering post_write_hook fn: 1
post_write_hook: tagmap_clrn address: 95817000 size: 35
 ++ syscall: 4
 .........entering pre_write_hook fn: 1
 ---> pre_write_hook: tagmap_issetn test - Address: 95817000 Size: 14
 .........entering post_write_hook fn: 1
post_write_hook: tagmap_clrn address: 95817000 size: 14
 ++ syscall: 4
 .........entering pre_write_hook fn: 1
 ---> pre_write_hook: tagmap_issetn test - Address: 95817000 Size: 15
 .........entering post_write_hook fn: 1
post_write_hook: tagmap_clrn address: 95817000 size: 15
 ++ syscall: 4
 .........entering pre_write_hook fn: 1
 ---> pre_write_hook: tagmap_issetn test - Address: 95817000 Size: 13
 .........entering post_write_hook fn: 1
post_write_hook: tagmap_clrn address: 95817000 size: 13
 ++ syscall: 4
 .........entering pre_write_hook fn: 1
 ---> pre_write_hook: tagmap_issetn test - Address: 95817000 Size: 10
 .........entering post_write_hook fn: 1
post_write_hook: tagmap_clrn address: 95817000 size: 10
 ++ syscall: 197
 ......sysexit_save call to tagmap_clrn
 +++++++++ sys call id: 197  addr: bf9a1280 size: 96
 ++ syscall: 192
 ++ syscall: 4
 .........entering pre_write_hook fn: 1
 ---> pre_write_hook: tagmap_issetn test - Address: 95817000 Size: 1
 .........entering post_write_hook fn: 1
post_write_hook: tagmap_clrn address: 95817000 size: 1
 ++ syscall: 3
 ......post_read_hook fn: 0
```

```
 post_read_hook: not an interesting source - clear tags
 .....post_read_hook: tagmap_clrn addr: 957f8000 size: 2
 ++ syscall: 197
 ......sysexit_save call to tagmap_clrn
 +++++++++ sys call id: 197  addr: bf9a11f0 size: 96
 ++ syscall: 192
 ++ syscall: 3
 ......post_read_hook fn: 5
 .....post_read_hook: tagmap_setn addr: 957ec000 size: 402
  ***** Tagged data verified
 ++ syscall: 4
 .........entering pre_write_hook fn: 1
 ---> pre_write_hook: tagmap_issetn test - Address: 95817000 Size: 72
 ***** Tagged data being written at Address 95817000 Size: 72
 ********* ALERT!! ***********
 .........entering post_write_hook fn: 1
post_write_hook: tagmap_clrn address: 95817000 size: 72
 ++ syscall: 197
 ......sysexit_save call to tagmap_clrn
 +++++++++ sys call id: 197  addr: bf9a1280 size: 96
 ++ syscall: 192
 ++ syscall: 4
 .........entering pre_write_hook fn: 1
 ---> pre_write_hook: tagmap_issetn test - Address: 95817000 Size: 1
 .........entering post_write_hook fn: 1
post_write_hook: tagmap_clrn address: 95817000 size: 1
 ++ syscall: 4
 .........entering pre_write_hook fn: 1
 ---> pre_write_hook: tagmap_issetn test - Address: 95817000 Size: 37
 .........entering post_write_hook fn: 1
post_write_hook: tagmap_clrn address: 95817000 size: 37
 ++ syscall: 4
 .........entering pre_write_hook fn: 1
 ---> pre_write_hook: tagmap_issetn test - Address: 95817000 Size: 35
 .........entering post_write_hook fn: 1
post_write_hook: tagmap_clrn address: 95817000 size: 35
 ++ syscall: 4
 .........entering pre_write_hook fn: 1
 ---> pre_write_hook: tagmap_issetn test - Address: 95817000 Size: 14
 .........entering post_write_hook fn: 1
post_write_hook: tagmap_clrn address: 95817000 size: 14
 ++ syscall: 4
 .........entering pre_write_hook fn: 1
 ---> pre_write_hook: tagmap_issetn test - Address: 95817000 Size: 15
 .........entering post_write_hook fn: 1
post_write_hook: tagmap_clrn address: 95817000 size: 15
 ++ syscall: 4
 .........entering pre_write_hook fn: 1
 ---> pre_write_hook: tagmap_issetn test - Address: 95817000 Size: 13
 .........entering post_write_hook fn: 1
post_write_hook: tagmap_clrn address: 95817000 size: 13
 ++ syscall: 4
 .........entering pre_write_hook fn: 1
 ---> pre_write_hook: tagmap_issetn test - Address: 95817000 Size: 10
 .........entering post_write_hook fn: 1
```

```
159  post_write_hook: tagmap_clrn address: 95817000 size: 10
160   ++ syscall: 4
161   .........entering pre_write_hook fn: 1
162   ---> pre_write_hook: tagmap_issetn test - Address: 95817000 Size: 1
163   .........entering post_write_hook fn: 1
164  post_write_hook: tagmap_clrn address: 95817000 size: 1
165   ++ syscall: 3
166   ......post_read_hook fn: 0
167   post_read_hook: not an interesting source - clear tags
168   .....post_read_hook: tagmap_clrn addr: 957f8000 size: 2
169   ++ syscall: 6
170   --------- post_close_hook: 4
171   ++ syscall: 6
172   --------- post_close_hook: 5
173   ++ syscall: 91
174   ++ syscall: 4
175   .........entering pre_write_hook fn: 6
176   ---> pre_write_hook: tagmap_issetn test - Address: 95780000 Size: 72
177   ***** Tagged data being written at Address 95780000 Size: 72
178   ********* ALERT!! ***********
179   .........entering post_write_hook fn: 6
180  post_write_hook: tagmap_clrn address: 95780000 size: 72
181   ++ syscall: 6
182   --------- post_close_hook: 6
183   ++ syscall: 91
184   ++ syscall: 252
```

While this logfile resembles the one shown in Appendix A, this one has entries that were omitted from the previous example to focus on the tagging activities. All entries are included here for a more comprehensive view.

- The first two lines are associated with the initialization of Pin and libdft-um Pintool. Line 2 logs the allocation of memory for the tagmap that is used to track the tag status of the remaining virtual machine address space.
- Lines 4 through 34 are documenting system calls that are used by the operating system to allocate resources and start the execution of the application.

- Lines 35 -41 are associated with the creation of a network socket and binding of the socket to a local address. The network socket has been assigned the file designator – 4.
- Lines 42 and 43 are related to the dynamic memory allocation process that is moving the heap break point.
- Line 45 is one of the more important events in this experiment, as it indicates that a data file has been opened and has been assigned the file designator – 5.
- Line 46 indicates that the file name was passed to the enforcement mechanism, which will then be used to identify the associated license file.
- Line 47 indicates that an associated license file has been opened and is being parsed.
- Line 48 indicates that in the license file, the sensitivity_level parameter specifies that the data are classified as sensitive. As a result, any data read from the file will be tagged.
- Lines 49 – 53 show a similar open process for the file test_data2.txt. In this case, the file designator is 6, and the license specifies that data in the file are non-sensitive.
- Lines 54 – 57 are operating system operations associated with checking file status and memory management.
- Lines 58 – 105 represent the actions that are associated with presenting the user the option table, which is shown in the above user dialog, and the

user selecting an input option. (This is the same as what was shown previously in Appendix A.)

- Lines 106 – 109 are operating system operations associated with checking file status and memory management.
- Lines 110 – 113 show that data are being read from the input file, which has already been designated as sensitive, and that the data tagging has been verified. For efficiency, the operating system has read in more data than what was requested in the application source code and all data are contained in the data buffer at address 0x957ec000.
- Lines 114 – 120 show that one line of the data read in from the sensitive data file are being written out to stdout (file descriptor 1). Writes to stdout are done immediately, rather than buffering outputs as is done with block devices, such as disk data files. In file read operation, 402 bytes were input. To write the first line to stdout, only 72 bytes were transferred back out. Before the data are actually output, the associated tags are checked and as shown, the enforcement mechanism has correctly detected a write to a non-sensitive output.
- Lines 121 – 124 are operating system operations associated with checking file status and memory management.
- Lines 125 – 168 again show the operations associated with presenting the user with input options and the user selecting an input source. In this case, the user selects the Quit option.
- Lines 169-173 show the closing of the network socket and input file.

- Lines 174 – 180 show the one line that has been read in from file 1 now being written out to file 2. The write is delayed as the operating system will combine a sequence of writes operation into one for block type devices, such as disk files. As only one line is to be written, it is performed now. The enforcement mechanism checks the data and detects that tagged data are being written and generates an alert.
- Lines 181 – 182 show that the second data is being closed.
- Lines 183-184 show the termination of the process.

These results demonstrates that with an ability to parse the contents of a license file that specifies the usage policies, the usage management enforcement mechanism can automatically determine which data sources are considered sensitive and can tag and track the flows of data from those sources as they propagate through an unmodified application.

Appendix C – Enforcement Mechanism Software Excerpts

The Usage Management (UM) enforcement mechanism software developed in this research is based on the Pin instrumentation software and the libdft software developed for taint analysis (Luk, et al., 2005) (Kemerlis, Portokalidis, Jee, & Keromytis, 2012). This appendix describes the major components of the enforcement mechanism software and illustrates how the collection of software components performs instrumentation, data flow tracking (DFT), license parsing, and usage management enforcement.

The software consists of a number of major components or modules. The first component is *Pin*, which is a software instrumentation tool developed by Intel that supports multiple operating systems and processor architectures. Pin provides the essential capabilities needed to monitor application execution in order to enforce how data are used. The instrumentation capabilities are selectively applied and controlled by a program that is called a *Pintool*.

The main module that initiates the data flow tracking and UM enforcement is a c module called *libdft-um.c*. This module, which is described in more detail below, initiates the Pin instrumentation, includes specific system call actions, processes usage policy licenses, and monitors attempts to output sensitive data.

The module *libdft_api.c* contains the DFT initialization code that directs Pin to instrument every assembly language instruction using the TRACE_AddInstrumentFunction(). It also includes capabilities to track multiple threads and store data when system calls are entered or exited. This module also includes integer mappings for the 8, 16, and 32 bit register references.

The module *tagmap.c* contains all of the code which: allocates tag memory for monitoring the entire 4MB processor address space, sets tags, retrieves tags, clears tags, and tests if tags are set for individual memory addresses or for blocks of addresses. The tagging software uses one bit to store a tag for each byte of memory space, so every operation requires a mapping from the byte(s) address of interest to the corresponding bit(s) in the tag map.

The module *syscall.c* contains a table describing system call characteristics including: number of arguments, the flag specifying if arguments should be saved on entry, the flag specifying if return values should be saved, an arguments map, the pre-syscall routine to be executed, and the post-syscall routine to be executed. In addition, the module contains some syscall instrumentation routines, some of which are redundant and are not used.

The module *libdft_core.c* includes the function *ins_inspect()* in which every assembly language instruction is evaluated and when appropriate, propagates the tags. For example, if the instruction is an ADD and one operand is tagged while the second is not, the sum must be tagged; the tag is propagated to the sum. The module contains a number of functions to deal with the variety of operand types and addressing modes. The module must consider every instruction for a potential flow of tagged data, so it is quite large. (The module currently does not process any floating point instructions.)

libdft-um

To illustrate the UM enforcement mechanism software operation, key components of the libdft-um.c module are described below.

```
/*
 * libdft-um
 *
 * a tool for enforcing usage management by monitoring all
 * data flows from sources identified as sensitive.  when
 * attempts are made to write sensitive data, the instrumentation
 * will generate an alert.  The alert can be used as a basis for
 * prevent the an actual write to sdtout, a data file, a network
 * socket, or a pipe.
 */
int
main(int argc, char **argv)
{
	/* initialize symbol processing */
	PIN_InitSymbols();

	/* initialize Pin; optimized branch */
	if (unlikely(PIN_Init(argc, argv)))
		/* Pin initialization failed */
		goto err;

	/* initialize the core tagging engine */
	if (unlikely(libdft_init() != 0))
		/* failed */
		goto err;



	/* Instrument System calls of interest   */

	/* read(2) */
	(void)syscall_set_post(&syscall_desc[__NR_read], post_read_hook);

	/* readv(2) */
	(void)syscall_set_post(&syscall_desc[__NR_readv], post_readv_hook);

	/* write(2) */
	(void)syscall_set_pre(&syscall_desc[__NR_write],pre_write_hook);
	// NEW FUNCTION
	(void)syscall_set_post(&syscall_desc[__NR_write], post_write_hook);



	/* socket(2), accept(2), recv(2), recvfrom(2), recvmsg(2)     */
	/* send(2), sendto(2), sendmsg(2)
	*/
	if (net.Value() != 0) {
		(void)syscall_set_pre(&syscall_desc[__NR_socketcall],
			pre_socketcall_hook);
		(void)syscall_set_post(&syscall_desc[__NR_socketcall],
```

```
                post_socketcall_hook);
                }

        /* dup(2), dup2(2) */
        (void)syscall_set_post(&syscall_desc[__NR_dup], post_dup_hook);
        (void)syscall_set_post(&syscall_desc[__NR_dup2], post_dup_hook);

        /* close(2) */
        (void)syscall_set_post(&syscall_desc[__NR_close], post_close_hook);

        /* open(2), creat(2) */
        /* use post_open_hook calls as fds are needed for tagging     */

        if (fs.Value() != 0) {
                (void)syscall_set_post(&syscall_desc[__NR_open],
                                post_open_hook);
                (void)syscall_set_post(&syscall_desc[__NR_creat],
                                post_open_hook);
        }

        /* add stdin to the interesting descriptors set */
        if (sin.Value() != 0)
                fdset.insert(STDIN_FILENO);

        /* Initialize xml library and check for version mismatches    */

        LIBXML_TEST_VERSION

        /* start Pin */
        PIN_StartProgram();

        /* typically not reached; make the compiler happy */
        return EXIT_SUCCESS;

err:    /* error handling */

        /* detach from the process */
        libdft_die();

        /* return */
        return EXIT_FAILURE;
}
```

The first section of *main()* initializes the Pin instrumentation, first by initializing the symbols and then the Pin software. Next, the libdft software is initialized using the libdft_init() function. In the libdft_init() function, the Pin software is configured to inspect assembly language instructions.

After the initialization, Pin is configured to instrument specific system calls. First, the read and readv system calls are instrumented after the functions are

executed, so that the enforcement mechanism can determine how many bytes have been read. This allows the mechanism to tag all of the bytes that are read from a sensitive source. Then, Pin is configured to instrument the write system calls both before and after execution. Before the write function is executed, the enforcement mechanism can check to see if any of the data to be written are tagged, allowing it to prevent any unwanted operations from actually taking place. Instrumenting the write after the operation permits the enforcement mechanism to clear the output buffer after a write, along with the corresponding tag bits.

The next section is used to configure the instrumentation of the network communications. A single socketcall function is used for multiple purposes, so instrumentation before and after execution is necessary because the function is used for both read and write operations. The instrumentation strategy used for the socket read and write operations is similar to the one used for the read and write system calls. In this version, network sockets can be specified as non-sensitive using an optional input parameter when the libdft-um program is started.

A file descriptor can be duplicated, resulting in multiple references to a data source or destination. The duplication is done using dup or dp2 system calls. The enforcement mechanism adds the duplicate of the sensitive file descriptor to the list of those that are monitored.

Next, when a file is closed, the operating system removes its file descriptor from the open file descriptors table and similarly, the enforcement

mechanism also removes the file descriptor from the monitored list. Any data that have been read from this file that still reside in memory retain their tags.

For this proof of concept research, the open and create system call instrumentation is significant because it demonstrates the capability that is essential for automatic enforcement. (With this version of code, the file open and create instrumentation can also be disabled by using an optional input parameter when the libdft-um program is started.) The details of the open system call instrumentation are described below.

Next, the standard interfaces can be designated as sensitive *(stdin, stdout, and stderr)* by including an input parameter when starting libdft-um. Ultimately, this would be specified in a policy license. When designated as sensitive, any data input from stdin is tagged.

The enforcement mechanism uses the libxml2 library and it must be initialized before use. This is done using the LIBXML_TEST_VERSION function.

After all of the initialization and specification of the instrumentation to be applied, then the execution is started using the PIN_StartProgram() function call. Now, we examine the characteristics of the major system call instrumentation routines.

post_open_hook

The post_open_hook() routine plays a key role in the automatic operation of the enforcement mechanism; it is executed immediately after the open system call is executed so that the file descriptor for the newly opened file is defined. The key operations are described below.

```
/*
 *
 * whenever open(2)/creat(2) is invoked,
 * add the descriptor inside the monitored
 * set of descriptors if the licenses dictate so
 * or, if no license exists also add to list.
 *
 *
 * NOTE: it does not track dynamic shared
 * libraries
 */
static void
post_open_hook(syscall_ctx_t *ctx)
{
    const char *pattern = SENSITIVITY;    // xml file pattern of interest
    const char *sens_string = SENSITIVE;  // xml string for sensitive info
    xmlDocPtr xmldoc;
    xmlChar * xml_string;   // pointer to pointer of xmldoc string in memory
    char * xml_substring;                 // pointer to second line of xmldoc
    int xml_string_size;

    char str_ret[10];                     //DIAGNOSTIC
    char str_filename[128];               //Used for license filename
    char str_flags[10];
    int num_chars_diff;                   // used for xml string compare

    sprintf(str_ret, "%d", ctx->ret);     //DIAGNOSTIC - fd of opened file
    sprintf(str_flags, "%d",ctx->arg[SYSCALL_ARG1]);

    /* not successful; optimized branch */
    if (unlikely((long)ctx->ret < 0))
        return;

    /* ignore dynamic shared libraries */
    if (strstr((char *)ctx->arg[SYSCALL_ARG0], DLIB_SUFF) == NULL &&
        strstr((char *)ctx->arg[SYSCALL_ARG0], DLIB_SUFF_ALT) == NULL)
        {

          /* determine if a license is available */

        LOG("..... post_open_hook - fn: " + string(str_ret) + "\n");
        strncpy(str_filename, (char *)ctx->arg[SYSCALL_ARG0],120);
        LOG("......opened filename: " + string(str_filename) + "\n");

        /* -- generate license file name using opened filename--      */
        strcat(str_filename, ".lic");

        /* extract the portion of license dealing with sensitivity    */
        xmldoc = extractFile(str_filename, (const xmlChar *)pattern);

        if(xmldoc != NULL) {
                    LOG(".......examining license file: " + string(str_filename) +
                    "\n");
            /* copy xmldoc to string in memory */
            xmlDocDumpMemory(xmldoc, &xml_string, &xml_string_size);
```

```
            /* set pointer to first delimiter '<' */
            xml_substring = strchr((char *)xml_string, '<');

            /* set point to second delimeter '<' */
            xml_substring = strchr(xml_substring + 1, '<');

            /* compare with reference to determine sensitivity */
            num_chars_diff = strncmp(xml_substring, sens_string, 50);

            if(num_chars_diff == 0) {
                LOG(".......file - " + string(str_filename) +
                " - is sensitive \n");

                fdset.insert((int)ctx->ret);      // Add to interesting file list
            }
            else {

                LOG(".......file - " + string(str_filename) +
                " - is not sensitive \n");
            }
        /* housekeeping here for xml functions */
        xmlFreeDoc(xmldoc);
        }
        else {

            /* case of file with no license - treat as sensitive */
            fdset.insert((int)ctx->ret);          // Add to interesting file list

        }
    }
}
```

After initialization and extraction of the system call parameters, the routine uses the name of the file just opened to form the corresponding license filename. Then, the routine attempts to open the license file and extract the specific XML file entry pointed to by the pattern pointer. This operation generates a subdocument, which is then parsed to determine the sensitivity value. If no valid license file is available, then the file is considered sensitive.

post_read_hook

The post_open_hook() routine is where data input from files are tagged. The data are tagged if the file being read is one that has been designated as sensitive. The key operations are described below.

```
/*
 * The read system call will be one potential source
 * of data that must be controlled by the usage management
 * enforcement mechanism.  Policy data associated with the
 * data read in will be used to determine whether or not usage
 * management enforcement is necessary.
 *
 * read(2) handler (tagged data - source)
 */
static void
post_read_hook(syscall_ctx_t *ctx)
{
    char str_arg0[10], str_arg1[20], str_ret[10];          //DIAGNOSTIC
        sprintf(str_arg0, "%d", ctx->arg[SYSCALL_ARG0]);   //DIAGNOSTIC
        sprintf(str_arg1, "%x", ctx->arg[SYSCALL_ARG1]);   //DIAGNOSTIC
        sprintf(str_ret, "%d", ctx->ret);                  //DIAGNOSTIC

  /* read() was not successful; optimized branch */
  if (unlikely((long)ctx->ret <= 0))
    return;

  /* tagged data source */
  /* Is the file one of the interesting data sources being tracked?  */

  LOG("......post_read_hook fn: " + string(str_arg0) + " \n");//DIAGNOSTIC

  if (fdset.find(ctx->arg[SYSCALL_ARG0]) != fdset.end())
    {
      LOG(".....post_read_hook: tagmap_setn addr: " + string(str_arg1) +
         " size: " + string(str_ret) + " \n");                    // Diagnostic

      /* set the tag bits */
      tagmap_setn(ctx->arg[SYSCALL_ARG1], (size_t)ctx->ret);

      /* diagnostic to verify tag bits are set */

      if (tagmap_issetn(ctx->arg[SYSCALL_ARG1], (size_t)ctx->ret) != 0)
       LOG(" ***** Tagged data verified \n");
    }
  else
    {
      LOG("\t post_read_hook: not an interesting source - clear tags \n");  //
Diagnostic
      LOG(".....post_read_hook: tagmap_clrn addr: " + string(str_arg1) +
         " size: " + string(str_ret) + " \n");                // Diagnostic

      /* clear the tag markings */
```

```
      tagmap_clrn(ctx->arg[SYSCALL_ARG1], (size_t)ctx->ret);
    }
}
```

Much of the code of the post_read_hook() routine is for diagnostics. The function uses system call parameters: file descriptor and buffer address, and the return value: number of bytes read, to set tag bits associated with the buffer addresses. If the file is one that is deemed sensitive, the bits are set, otherwise, they are cleared. The clear operation is necessary for application software that is opening and closing files on a repetitive basis, as the memory used for buffers will be reallocated and if there is a mix of sensitive and non-sensitive data, then non-sensitive data will be tagged as sensitive.

pre_write_hook

For effective control of sensitive data, the enforcement action must be performed before a write takes place. The basic operation is described below.

```
/*
 * This is a function that is used to detect and prevent unauthorized
 * transmission of data that must be controlled, in order to enforce
 * usage policies.  This instrumentation function is executed before the
 * actual write takes place.
 *
 * write(2) handler (tests for tagged data ) BEFORE EXECUTION !!
 */
static void
pre_write_hook(syscall_ctx_t *ctx)
  {

    char str_arg0[10],str_arg1[20], str_arg2[10];      //DIAGNOSTIC
    sprintf(str_arg0, "%d", ctx->arg[SYSCALL_ARG0]);   //DIAGNOSTIC
    sprintf(str_arg1, "%x", ctx->arg[SYSCALL_ARG1]);   //DIAGNOSTIC
    sprintf(str_arg2, "%d", ctx->arg[SYSCALL_ARG2]);   //DIAGNOSTIC

    LOG(".........entering pre_write_hook fn: " + string(str_arg0) +"\n");
    LOG("---> pre_write_hook: tagmap_issetn test - Address: " +
      string(str_arg1) + " Size: " + string(str_arg2) + "\n");//DIAGNOSTIC

    /* check the tag markings */

    if (tagmap_issetn(ctx->arg[SYSCALL_ARG1], ctx->arg[SYSCALL_ARG2]) != 0) {
       LOG("***** Tagged data being written at Address " + string(str_arg1) +
       " Size: " + string(str_arg2) + "\n");
```

```
//     tagmap_setn(ctx->arg[SYSCALL_ARG1], ctx->arg[SYSCALL_ARG2]);
                          // if some set, set all
       LOG("********* ALERT!! ***********\n");

       /* A FORCED RETURN FROM THE SYSTEM CALL WOULD GO HERE */

    }
}
```

As in the post_read_hook() routine, most of the code in the pre_write_hook() routine is for post-experiment verification of proper operation. The basic function that this routine does is check the *entire* content of the output buffer to determine if any tag bits are set. This approach is used to ensure that sensitive data are not encapsulated with non-sensitive data. A more conservative approach would be to tag all of the buffer contents as sensitive, if any of the contents are tagged. Once tagged data are detected, a forced return from the system call could be inserted as shown, thus preventing an unauthorized write.

_socketcall_hook

The enforcement mechanism uses two instrumentation routines: pre_socketcall_hook() and post_socketcall_hook(), in a manner similar to the pre_write_hook() and post_read_hook(). As mentioned before, a single socketcall is used for a number of functions, which are specified by one of the parameters when calling the routine. When data are to be sent out through the network socket, they must be checked before the send takes place, so the pre_socketcall_hook() routine takes action if a send, sendmsg, or sendto operation is requested. The post_socketcall_hook() routine will tag incoming data for recv, recvmsg, or recvfrom operations. The post_socketcall_hook also performs initialization functions, similar to the post_open_hook() routine.

These descriptions provide a high-level view of the usage management enforcement mechanism operations. To summarize, the concept of operations is to use licenses to specify usage policy for data and encapsulate an application with instrumentation software to enforce the policies. The enforcement is done by tagging data that requires control, tracking the flow of the data, and then permitting or denying the flow of information to other destinations.

Bibliography

Badger, L., Grance, T., Patt-Comer, R., & Voas, J. (2011). *DRAFT The NIST Definition of Cloud Computing, Special Publication 800-145.* Gaithersburg, MD: National Institute of Standards and Technology.

Bishop, M. (2003). *Computer Security: Art and Science.* Upper Saddle River, NJ: Pearson Education, Inc.

Chen, X., Garfinkel, T., Lewis, E. C., & Subrahmanyam, P. (2008). Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. *ASPLOS XIII Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (pp. 2-13). ACM.

D. Elliott Bell, L. J. (1973). *Secure Computer Systems: Mathematical Foundations ESD-TR-73-278.* , Electronic Systems Division,. Hanscom AFB, Bedford Massachusetts: Air Force Systems Command.

Fenton, J. (1974). Memoryless Subsystems. *The Computer Journal*, 143-147.

Foley, S. N. (1989). A Model for Secure Information Flow. *Proceedings of the 1989 IEEE Symposium on Research in Security and Privacy* (pp. 248-258). IEEE.

Gentry, C. (2010). Computing Arbitrary Functions of Encrypted Data. *Communications of the ACM*, 97-105.

Jamkhedkar, P. A., Heileman, G. L., & Lamb, C. C. (2010). An Interoperable Usage Management Framework. *DRM '10 Proceedings of the Tenth Annual ACM Workshop on Digital Rights Management* (pp. 73-88). ACM.

Jamkhedkar, P. A., Lamb, C. C., & Heileman, G. L. (2011). Usage Management in Cloud Computing. *2011 IEEE International Conference on Cloud Computing (CLOUD)* (pp. 525-532). IEEE.

Kemerlis, V. P., Portokalidis, G., Jee, K., & Keromytis, A. D. (2012). libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. *VEE '12 Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments* (pp. 121-132). ACM.

King, S., & Chen, P. (2006). SubVirt: Implementing Malware with Virtual Machines. *IEEE Symposium on Security and Privacy* (p. 14pp). Oakland: IEEE.

LaPadula, L. J., & Bell, D. E. (1996). MITRE Technical Report 2547, Volume II. *Journal of Computer Security*, 239-263.

Lauter, K., Naehrig, M., & Vaikuntanathan, V. (2011). Can Homomorphic Encryption Be Practical? *CCSW '11 Proceedings of the 3rd ACM Workshop on Cloud Computing Security.* ACM.

Liu, F., Mao, J., Bohn, R., Messina, J., Badger, L., & Leaf, D. (2011). *NIST Cloud Computing Reference Architecture, Special Publication 500-292.* Gaithersburg, MD: National Institute of Sandards and Technology.

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lwoney, G., . . . Hazelwood, K. (2005). Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *Proceedings of the 2005 PLDI* (pp. 190-200). ACM.

Nandina, V., Luna, J.-M., Nava, E. J., Lamb, C. C., Heileman, G. L., & Abdallah, C. T. (2013). Policy-based Security Provisioning and Performance Control in the Cloud. *CLOSER 2013 - Proceedings of the 3rd International Conference on Cloud Computing and Services Science* (pp. 502-508). SciTePress.

Olympics.org. (2014). *Sochi Bids Farwell*. Retrieved from Olympics.org: http://www.olympic.org/sochi-2014-winter-olympics

Paar, C., & Pelzl, J. (2010). *Understanding Cryptography A Textbook for Students and Practicioners.* Heidelberg: Springer-Verlag.

Pappas, V., Kemerlisl, V. P., Zavou, A., Polychronakis, M., & Keromytis, A. D. (2012). *CloudFence: Enabling Users to Audit the Use of Their Cloud-Resident Data.* New York, NY: Columbia University Computer Science Technical Reports.

Park, J., & Sandhu, R. (2004, February). The UCON ABC Usage Control Model. *ACM Transactions on Information and System Security*, pp. 128-174.

Richelson, J. T. (1995). *A Century of Spies, Intelligence in the Twentieth Century.* New York. NY: Oxford University Press.

Schneier, B. S. (1996). *Applied Cryptography.* New York, NY: John Wiley & Sons, Inc.

TCG. (2011). *Trusted Platform Module Main Specification Level 2 Version 1.2, Revision 116.* Trusted Computing Group.

Veilard, D. (2014, December 5). *The XML C parser and toolkit of Gnome*. Retrieved from The XML C parser and toolkit of Gnome: http://xmlsoft.org/index.html

Zhang, F., Chen, J., Chen, H., & Zang, B. (2011). CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. *SOSP '11 Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (pp. 203-216). ACM.