A virtualized usage control bus system*

Cornelius Moucha†
Fraunhofer IESE, Information Systems Quality (ISQ)
Kaiserslautern, Germany
cornelius.moucha@iese.fraunhofer.de

Enrico Lovat, Alexander Pretschner
Karlsruhe Institute of Technology
Karlsruhe, Germany
lovat@kit.edu and pretschner@kit.edu

Abstract

Usage control is an extension of access control that additionally defines what must and must not happen to data after access has been granted. The process of enforcing usage control requirements on data must take into account all the different representations that the data may assume at different levels of abstraction (e.g. file, window content, network packet). Therefore, multiple data flow tracking and usage control enforcement monitors are likely to exist, one at each relevant layer. Whenever data flows from a representation at one layer to a representation at another layer (e.g. a file is loaded and interpreted by an application), then the monitor for the initiating layer (in the example, the operating system) must notify the monitor for the receiving layer (in this example, an application, like a browser) about the data being transfered. This is required in order to associate both representations to the same data. In this paper, we present a bus system to support system-wide usage control enforcement that, for security and performance reasons, is implemented in a hypervisor. We provide an example application for enforcing usage control across layers of abstraction in the context of social networks. We evaluate security and performance of our bus system.

Keywords: Data-flow tracking, usage control, bus system, virtualization, information flow.

1 Introduction

Data usage control [2, 3, 4] is concerned with requirements that pertain to handling data after access to it has been granted. Among other domains, usage control is relevant in the context of data protection, management of intellectual property, compliance with regulations and digital rights management.

Once data consumers have received the data, in our context together with a policy that stipulates usage control requirements, they use it. Using data means, for example, to render it on a screen, print it on paper, execute it (if it is executable), modify it (e.g. with a text editor or a spreadsheet application), copy and disseminate it or delete it. In case one or more of such usage actions takes place, data is potentially transformed into one or more additional representations.

If usage control requirements are to be enforced on *data*, then this likely means that they have to be enforced on *all representations of that data*. For instance, a picture can exist as a file, as content in a window, as a Java object, and as part of a text document.

In a setting where enforcement is done on all rather than just one initital data representation, all meaningful usage actions performed on the different representations must be tracked at (and across) the different system layers. We have previously implemented data flow tracking systems that attribute a semantics to the layer-specific actions. This semantics captures how data flows in-between different representations within one layer of abstraction [5]. At runtime we can, for instance, track the flow of data in-between memory regions, files or network sockets or, more generally, express how the mapping from data to representations is updated upon execution of a relevant action.

Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications, volume: 2, number: 4, pp. 84-101

*This paper is an extended version of the work originally presented at the 6th International Conference on Availability, Reliability and Security (ARES'11), Vienna, Austria, August 2011 [1].

†Corresponding author: Fraunhofer IESE, Information Systems Quality (ISQ), Fraunhofer-Platz 1, 67663 Kaiserslautern, Germany, Tel: +496316800-2111, Email: cornelius.moucha@iese.fraunhofer.de

One may argue that it is simpler to perform tracking and enforcement just at the level of machine code (processor instructions), instead of assuming one engine per layer. However, semantics of actions such as “copy” or “delete” differ depending on the layer of abstraction considered (copy&paste in a text editor, copy a file, clone an object, send an email), and it is almost impossible to automatically identify those parts of the machine code that correspond to such high-level usage actions, especially concerning today’s multicore or multiprocessor architectures. For this reason, the layered approach appears like a viable alternative. Moreover, conceptually (i.e. abstracting from technical details), such a layered approach scales well to distributed settings, like in the cloud domain, where the lack of centralized control makes usage control even more challenging.

Motivating Example As explained above, it is not sufficient to perform data flow tracking and related usage control enforcement at each layer in isolation. For the remainder of this work, we will refer to the following scenario as base example, illustrated in figure 1: consider a web browser that downloads a picture, e.g., from a social network. Once it has received the picture, the browser processes it, i.e., rendering on the screen. Furthermore, the Firefox monitor sends the policy attached to the picture to a policy storage using the bus communication (figure 1: arrow *1*). Afterwards, when the monitor intercepts an attempt of storing the picture in a cache file, the monitor for the operating system level is notified about the identity of data being transferred (arrow *2*) and only then the caching is allowed (arrow *3*). Hence, the data (the picture) exists in form of at least three different representations at the same time: as a set of pixel at the level of the window manager, as file at the level of operating system and as an object in the browser internal memory. Therefore, if the server ships the picture to the browser together with a policy, then such a policy must be propagated through the system and associated to each additional representation of the picture at every single layer. In general, for a system-wide enforcement of the policy, if the application initiates the creation of a representation R at a different layer L, then we must track the data flow that starts from R in L to all other representations at L and, recursively, to further layers. Moreover, on receiving the new data, the operating system monitor queries the policy storage to retrieve the policy associated to it (arrow *4*) and processes the answer (arrow *5*) enforcing it within its scope, e.g., prohibiting accesses to the cache file from applications others than Firefox itself.

Research Problem In sum, we tackle the problem of (1) how to communicate policies and (2) the flow of data from one layer of abstraction to another layer.

Solution Our solution consists of the design, implementation and evaluation of a bus system that we implemented at the hypervisor layer, both for security and performance reasons.

Contribution Representation-independent usage control [4] is a relatively new area of research. While several solutions have been proposed for the problem of the enforcement at single layers of abstraction [6, 5, 7, 8, 9, 10, 11], we are not aware of a generic implementation of cross-layer tracking and enforcement. Therefore, we see our contribution in the first architecture and implementation of a communication infrastructure for the enforcement of usage control requirements across layers of abstraction. This paper does not tackle issues related to conflicting policies [12], nor the problem of delegation [13]. This paper is an extended version of the conference contribution [1] providing a refined description of the communication infrastructure and resulting open issues as well as an extended evaluation.

Organization After a short overview of virtualization techniques, we illustrate our solution in Section 2 and contrast it with related work in Section 3. Section 4 lists detailed requirements while design choices

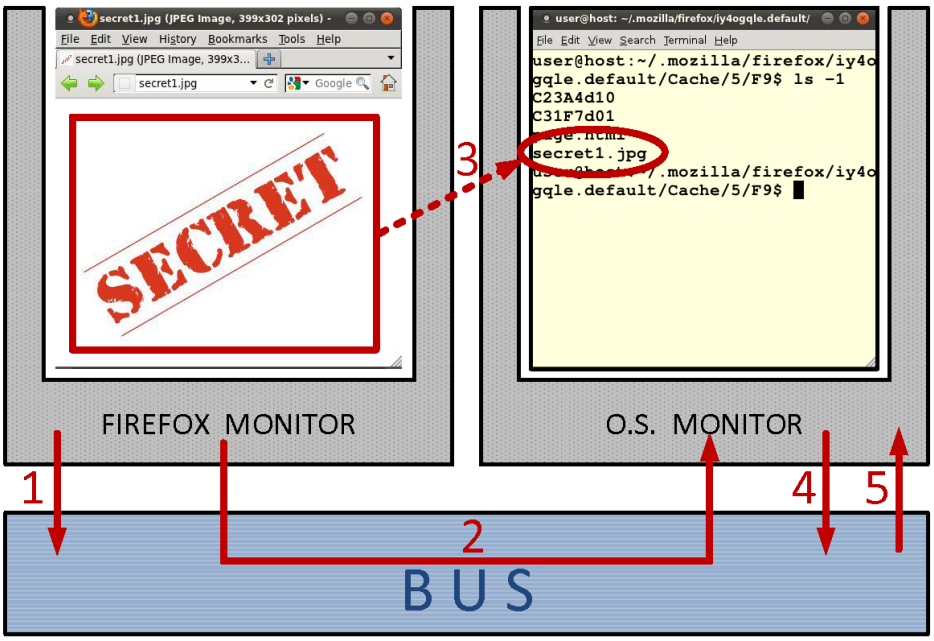

Figure 1: Example of cross-layer monitor communication using bus system

and implementation issues are presented in Sections 5 and 6. Evaluations of our system in terms of performance and security can be found in Section 7.

2 Background: Virtualization

While the basics of usage control have been introduced in Section 1, we recall the fundamentals of virtualization technology—on which our work builds due to the implementation of the bus in a hypervisor—in this section. We conclude this part by presenting the concrete infrastructure that we used to implement our system, the NOVA hypervisor.

Virtualization is a methodology of logically dividing computer resources of one physical machine, mainly hardware such as processor, main memory, network connectivity and others, into separate virtual machines executing their instructions independently of each other. The virtual machine monitor (VMM), often directly called *hypervisor*, is an application responsible for providing an environment in which several virtual machines ("guests") share the resources of one single host. This is achieved by controlling requests and responses for hardware resources. Hypervisors are divided in two categories, mostly differing in the capability of directly accessing hardware [14]. A *type-1 hypervisor* is a specialized system that runs directly on the host's hardware and that manages access requests from the guest OSs. In this configuration, only a minimal OS tailored for virtualization support exists between the host's hardware and guest system. I/O performance of the guest OSs is hence rather high. However, specialized hardware is required in order to support the additional hypervisor layer. A *type-2* (or hosted) hypervisor is a common user process running in a conventional fully-fledged OS. Therefore, its major application is in desktop virtualization solutions. Due to the required host OS, this solution usually does not perform as efficiently as the type-1 hypervisor but, on the other hand, does not pose any special constraint on the underlying hardware platform.

Virtualization techniques using type-1 hypervisors include paravirtualization, full virtualization and

hardware-assisted virtualization. Hardware-assisted virtualization is a platform virtualization approach that enables efficient full virtualization with help of hardware capabilities, primarily from the host processor. Paravirtualization requires adaptations of the guest kernels in order to use the hypervisor.

Because it is inappropriate to force data consumers to reinstall or rearrange their computer systems, our requirements for the bus infrastructure include the possibility of running the communication infrastructure for a generic preinstalled OS and with a minimal overhead concerning both, performance as well as disk usage. This precludes the use of a type-2 hypervisor, due to the decreased performance generated by the additional OS, and therefore requires the adoption of a full virtualization technique, considering the guest kernels' modifications needed by paravirtualization.

Recent processor generations provide hardware support for virtualization directly integrated in the CPU, namely Intel-VT (VT-x) or AMD-V. In addition to facilitating two different operation modes, for virtual machines and hypervisor respectively, these virtualization extensions offer an additional instruction, "VMCALL" [15], for switching the execution context from the guest OS to the associated virtual machine monitor. This particular instruction will be the key support for the communication between bus infrastructure (in the host environment) and monitors (in the guest system). Due to the hardware situation during development, we used an Intel processor and the corresponding VT-x virtualization extensions during design and implementation, but the same concepts and techniques are applicable to appropriate AMD processors.

One virtualization environment for the desired communication infrastructure is the NOVA OS Virtualization Architecture [16], a type-1 hypervisor solution. By applying microkernel construction principles in the area of full virtualization, NOVA bridges the gap between traditional microkernels and hypervisors to minimize the trusted computing base of the architecture [16]. Following the design criterion of component-based software engineering and the security principle of compartmentalization, NOVA consists of three functional separated parts: the microhypervisor itself, the root partition manager (called Sigma0) and the Virtual Machine Monitor (VMM). To minimize the attack surface, one instance of the VMM is responsible for just one virtual machine. Each of these components, as well as device drivers for bridging physical and virtual devices, additionally follows the principle of least privilege for maintaining a high level of security. NOVA provides a suitable framework for our work, mainly due to its architectural design and in particular to the component-based development and to its focus on security aspects. The entire communication infrastructure can be developed as a separate entity, requiring only minimal changes to the basic architecture. By using a type-1 hypervisor, no additional host OS is required, thus minimizing modifications and workload on the data consumer's system.

3 Related Work

Related work are cited throughout the text. Additionally, virtualization environments such as the introduced NOVA architecture usually provide the possibility for inter-domain communication between virtual machines. The Xen hypervisor uses XenBus for this connection [17]. Furthermore, several communication interfaces inside the OS exists. This includes common bus systems such as D-Bus [18] and generic inter-process communication like local sockets or shared memory. Several existing research projects introduce hypervisors dedicated to different specialized functionalities, like Bitvisor [19], Tiny Virtual Machine Monitor [20] or SecVisor [21]. The latter as a hypervisor dedicated to security functionality, especially harddisk encryption. Although all of the presented possibilities can be extended for solving our introduced problem, we are not aware of a solution providing an external bus system for a fast and secure communication between common applications inside the virtualized operating system.

4 Requirements

Our goal is to provide a secure and fast communication infrastructure for usage control-related message exchange across different levels of abstraction, illustrated in the introduction in figure 1. The infrastructure should also provide an interface for storing and retrieving data from/to a predefined secure location (requirement 2 below). Here *data* refers to both policies for enforcing usage control as well as data usage information. Data usage information, for instance, includes the number of times a file has already been accessed if a related policy restricts the number of accesses.

As explained in Section 2, the bus system has to be accessible independently of the data consumer's OS in which usage control monitors are executed, referred to as "platform independence of bus implementation" in requirement 1.1. Considering the amount and the extremely sensible nature of messages, security has to be considered as well, including tamper-resistant message exchange (requirement 4), secure storage location for policies and additional usage information (requirement 2), and high availability of the communication system (requirement 5) with as little performance overhead as possible (requirement 3). Data consumers usually have (OS-)administrator privileges and therefore can easily tamper in-memory solutions such as shared memory and terminate running applications and services, including existing bus systems. According to the security requirements this must not be possible in our solution. Instead of relying on existing solutions for bus systems or direct inter-process communication, we followed an approach that consists of "shifting" the usage-controlled system of the data consumer inside an virtualized environment that, in addition, provides the desired communication functionalities. Although tailored to usage control needs, the flexibility of the protocol design makes this solution easily extensible and suitable for other purposes.

As far as we know, bus systems out of the OS's scope for standard applications currently do not exist. In summary, we require a high level of security and performance, resulting in the following requirements:

1. Infrastructure for information exchange:
   - 1.1 Platform independence of bus implementation
   - 1.2 Minimal modifications to existing OS of the data consumer and virtualization architecture
   - 1.3 Virtualization of a preinstalled OS (data consumer's system) and extension to use communication functionalities
2. Shared, secure location for policy storage not modifiable from within the OS
3. Minimal performance overhead of communication infrastructure
4. Tamper-resistant message exchange
5. High availability of the communication system

5 Design

The NOVA architecture (see Section 2) provides several options for our communication infrastructure. One is to implement the bus component as a secure application. This is possible because in addition to hosting several guests, the NOVA architecture can host stand-alone applications which are executed independently of any VM and detached from any OS. Instead, they operate in parallel with the instantiated guest systems in an isolated execution environment, consequently including their own memory address space. These applications are executed at the same level as the partition manager and the VMM in the

hypervisor layer. Guest OSs inside VMs, operate in a special mode provided by the virtualization extensions and use the same protection ring as if they ran natively on the hardware. An effective isolation is thus achieved. The entire bus can be implemented in such a secure application, providing interfaces for the VMM associated with the usage-controlled VM.

Another option is to implement the bus component as a virtual device model, a purely software-based abstraction of hardware used by the guest systems to access existing physical devices. Virtual device models are usually a means to access a present physical correspondent of the emulated hardware; however, this connection to a physical entity is not mandatory. These device models are instantiated and managed by the VMM associated to the guest VM and can also be designed to access the bus component, without providing any device-specific functionality to the concerned guest OS. Basically, only a communication interface to the VMM is required.

Although only one guest OS is intended to be used in our context, separating the bus system from the concerned guest and the responsible VMM offers the opportunity to extend the solution across several VMs in future work. This scenario would provide one central bus authority for any registered usage control monitor in each VM.

However, sticking to the single-guest scenario addressed by our work, we decided to use the virtual device model, illustrated as *Device Drivers* in figure 2. The reason is that although a bus in a separate application provides stricter isolation, it also requires an additional step in the communication chain between usage control monitors and the bus component. This is due to the design of the message scheme of NOVA that does not allow direct communication between entities in its userland, i.e. between different VMMs or applications running at the hypervisor layer. Hence, the secure application solution results in necessary modifications of the NOVA architecture. In contrast, virtual device models are directly integrated into the VMM. Exemplary device model instantiations in figure 2 include the virtual process *VCPU*, a virtual harddisk *HDD* in addition to the developed bus component *UCbus*. By initiating the VMM, designated virtual devices are instantiated. Both communication partners, the VMM and the bus component *UCbus*, can exchange messages using already existing communication interfaces for device models. This minimizes the required modifications: we only need to provide a handler for the communication between guest OS and VMM, which is mandatory in both design scenarios.

Communication chain Figure 2 illustrates the required communication steps, indicated as arrows between involved components, for a usage control message (including a policy, a retrieval request for a policy, a usage event, the fact that data has been communicated from one layer to another, etc.). Once a communication has been initiated by a usage control monitor (the Firefox monitor in an exemplary Linux VM following the introduction's example), the first involved entity is the OS kernel; as shown in figure 2 the first communication step *1.* is from the usage control monitor to the kernel module. An additional component between the kernel and the monitor is a userspace library, only used for encapsulating the required communication facility between userspace and kernel, therefore not mentioned in the communication chain figure. The kernel module is responsible for translating the given memory address, which is relative to the monitor's virtual address space, into the kernel address space. A mapping between both address spaces is required as the VMM and, afterwards, the bus component *UCbus*, need to read the message's content (e.g., a policy or a message about the creation of a new representation) directly from the memory of the monitor. The message is then forwarded to the VMM (arrow *2.*) associated with the VM hosting the guest OS. Due to spatial memory isolation in NOVA for every instantiated VM, the prepared memory address of the kernel cannot be accessed natively, neither by the VMM nor by the bus component. For this purpose, this memory address has to be mapped into another appropriate address space of the VM itself. The *Partition manager* provides an interface for this translation (figure 2: arrow *3.*). Finally the last communication step is in the responsibility of the VMM to forward the message with

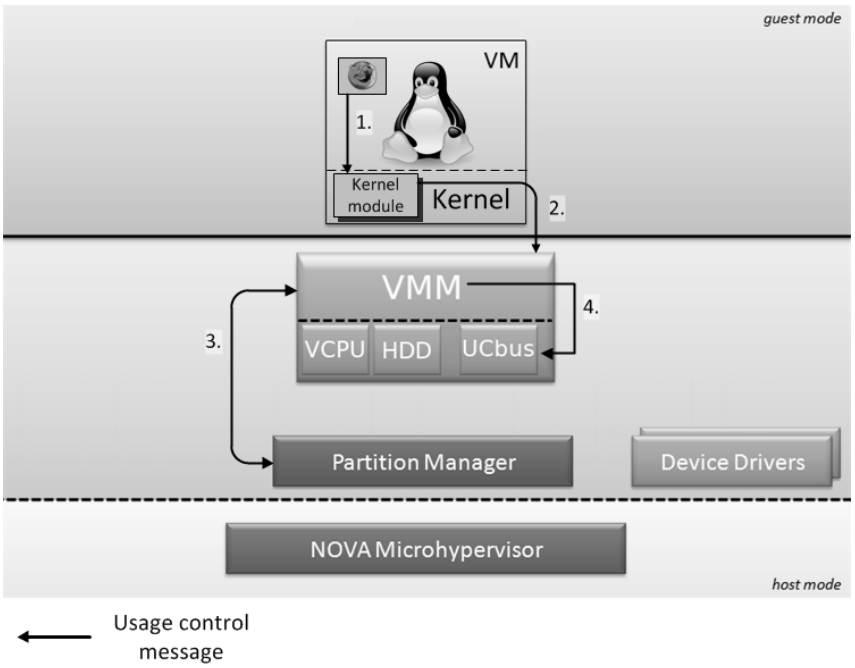

Figure 2: Communication chain: Messages to bus component

the translated memory address to the bus component (figure 2: arrow *4.*).

Here the message is processed and depending on the original purpose finally transmitted to the receiving entity. As shown in figure 3, for a notification the message is forwarded to another monitor (dotted arrow). Following the introduction's example this is the usage control monitor responsible for the operating system (OSm). For messages concerning policies or data usage information such as storing or retrieving, they are forwarded to a central policy storage (dashed line), which is also instantiated by the virtual machine monitor similar to the bus component itself.

Communication protocol In order to exchange usage control information, different types of messages have to be considered. First of all, to avoid malicious entities joining the communication, each usage control monitor must be registered to the communication infrastructure. During this registration, the monitor submits its observation interface and receives a unique monitor identifier. This observation interface identifies the layer of abstraction in which the monitor can enforce a given policy, e.g., file access, windowing system or Firefox application. In order to identify valid communication partners, a OS specific process identifier *PID* must also be submitted during the registration process. This authorization check is also discussed in section 7.2 (assumption A.2). Additionally, registration allows for later communication with monitors whose monitor ID is currently still unknown, by using the submitted observation interface as receiver. Moreover, there exists a second type of messages, used for storing and retrieving policies and usage information from the shared storage.

A notification message type has also to be provided: the receiver, identified either by a monitor ID or an observation interface ID, has to be notified about the presence of a message to be retrieved. This includes the notification to other monitors for enforcing a given policy or tracking the information flow. Because the communication from the bus to the guest system operates asynchronously via interrupts, the notification information has to be stored temporarily until the receiver monitor fetches the message upon receiving the IRQ. Finally, an interface for fetching a buffered message has to be provided. A summary of necessary message types is given in table 1.

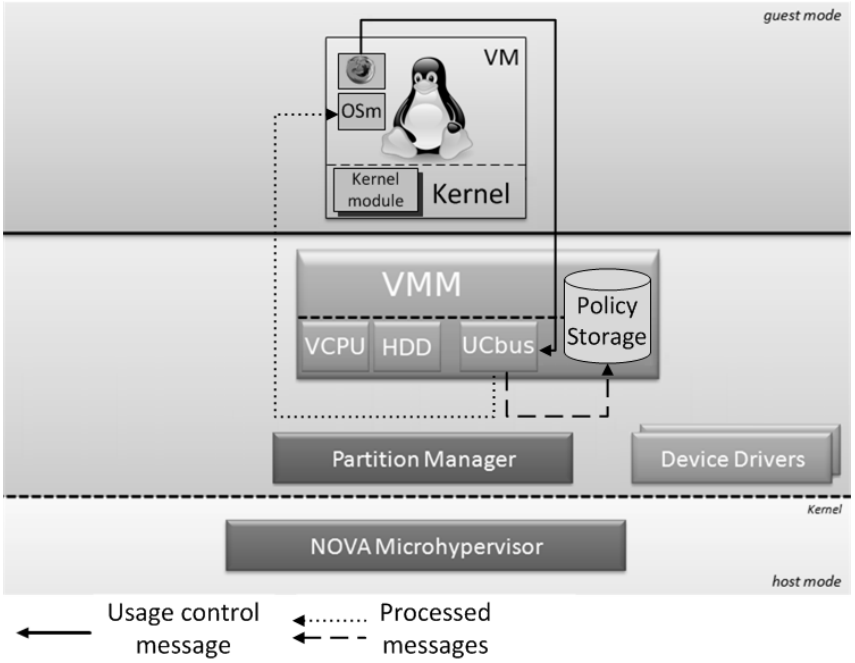

Figure 3: Communication chain: Processed messages

Message type	Description	Required parameters (with appropriate content type)
Registration	Register monitor to bus for validation of later communication attempts	Monitor observation interface (ID), process identifier (ID)
Store policy	Store policy in shared policy storage	Policy (memory address)
Get policy	Retrieve policy from shared policy storage	Policy or data item, receiver buffer (ID, memory address)
Notification	Notify another monitor for enforcing denoted policy or protecting given data item	Monitor or observation interface, policy or data item (ID, ID)
Fetch message	Fetch temporarily stored message from bus	Receiver buffer (memory address)

Table 1: Communication protocol: Message types

6 Implementation

After describing and motivating the architecture in the last section, we now sketch the implementation of the communication infrastructure. We consider each component involved in the communication chain together with its interface to other components.

The userspace library, the connector to the monitors, is designed as a common API library so that it can be used easily by monitor developers. It has to provide an interface for encapsulating the required communication with the kernel module for all introduced message types of the communication protocol. The kernel module, responsible for translating given memory references from the user address space into the physical address range of the virtualized OS, is implemented using the process filesystem *procFS* for communication with user applications. To separate the monitor registration from other message types,

one generic communication line is solely dedicated to registrations whereas another unique communication interface is prepared for each monitor after a successful registration. For the communication with the virtualization environment, the *VMCALL* instruction from the virtualization extensions is used to escape the execution context of the OS and jump to a predefined handler in the associated VMM. The inverse direction for communicating with the virtualized OS is implemented using interrupts. The kernel thus has to register an IRQ handler for this task. Following requirement 1.2, the only modification in the VMM is the handler method for receiving VMCALLs from the guest. For the required address translation, a message for accessing the guest's memory is sent to the partition manager Sigma0 using existing communication capabilities of NOVA. Finally, the usage control message is forwarded to the bus component, where it is processed according to the message type.

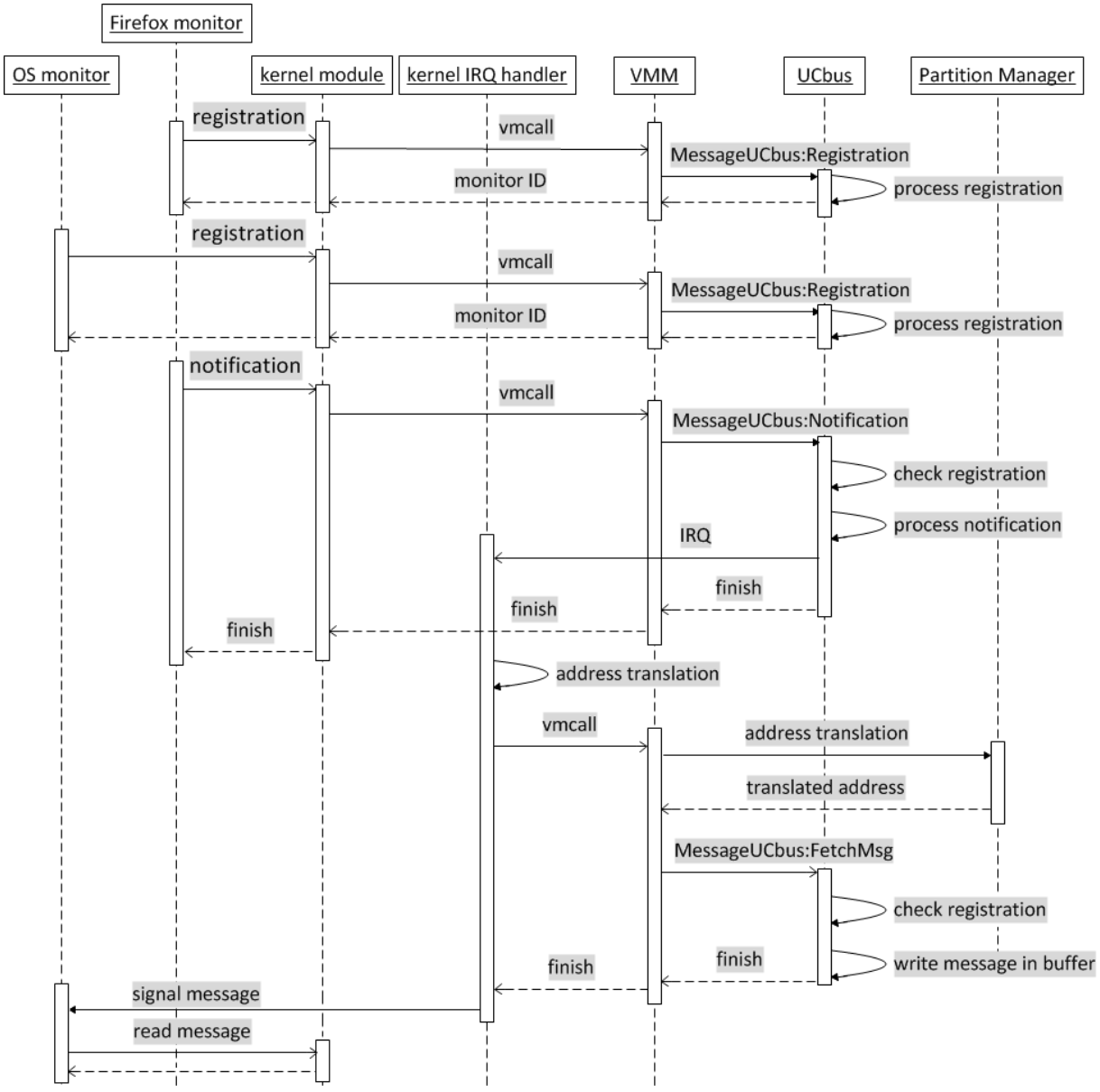

Figure 4: Notification to other monitors

Revisiting the example from the introduction, the notification behavior is illustrated in figure 4:

The monitors for Firefox and the OS first have to register with the bus. After receiving a web page with sensitive content including an appropriate usage control policy, the Firefox monitor automatically generates an appropriate policy for protecting the cache files from unauthorized reading, stipulating that any other application except Firefox itself is not allowed to read the cache entries. This policy is stored in the shared policy storage, and the OS-level receiver monitor, capable of controlling file access, is informed with a notification message. Afterwards, the OS monitor has to protect the cache file; this as well as the policy generation and storage procedures are not shown in figure 4, which only depicts the behavioral schema of registration and notification, respectively.

7 Evaluation

Requirement 1 (Section 4) and its sub-requirements are met by design: establishing the communication infrastructure in the virtualization environment provides the possibility to use the bus functionality regardless of the actual operating system of the data consumer. Due to the implementation of the bus component as virtual device model, the virtualization architecture was only modified for providing a handler for the *VMCALL* instruction, whereas the guest OS was not modified at all, but rather extended to support the communication interface. Such an extension only relies on those mechanisms provided by OSs to append additional functionalities, e.g. loading a kernel module. The NOVA architecture offers the opportunity to use a preinstalled operating system as guest VM, complying with requirement 1.3. In the following, we provide evidence that the other requirements are satisfied as well.

7.1 Performance

We evaluate the performance of the communication infrastructure, i.e., the message exchange between one monitor and the bus component, as well as the communication between two monitors using a notification message. To do so, we compare our solution to the software-based bus implementation D-Bus [18] providing the same functionality and message interface, and factor out the impact of the virtualization infrastructure by evaluating the following scenarios:

1. Debian Linux in NOVA: UCbus implementation
   This scenario consists of our solution with the bus component in the NOVA virtualization environment. The guest OS is a common Debian Lenny installed natively on a physical harddisk.

2. Debian Linux native: D-Bus implementation
   The guest OS mentioned in scenario 1 is booted as native OS without the virtualization environment. Using the system's D-Bus daemon, monitors can exchange messages using an interface similar to the solution of this paper, but via the D-Bus.

3. Debian Linux in NOVA: D-Bus
   To illustrate the impact of the virtualization environment, performance is evaluated for the same D-Bus implementation as introduced in scenario 2. Instead of using the OS natively, it is integrated into a virtual machine using NOVA.

For the performance evaluation, we considered message bundles instead of single messages. The introduced message types differ solely in the necessary processing in the bus component. Especially for storing and retrieving a policy, a memory copy operation is involved, whose execution time depends on the size of the actual policy. In general for sending a message to the bus, the communication chain is the same, whereas for a notification message also the rearward direction has to be considered. There, the destination monitor identifier and notification content has to be stored temporarily and subsequently

fetched by the receiver. In sum, for messages related to policy storage, a higher computational effort is required whereas for notifications the communication chain is more complex. In future application scenarios with different monitors exchanging messages, not only one message type is used but rather several. Therefore it is reasonable to combine several messages in a bundle for the evaluation. Hence, these message bundles consist of one message for storing and for retrieving a policy. Additionally, one notification message is included. Policy retrieval as well as notification messages have random values for the destination monitor and the concerned policy.

For each of the three scenarios, we measured the overall time consumption for the same evaluation procedure: After preparing the kernel module or accordingly initiating the D-Bus daemon, 15 simple applications processes acting as usage control monitors were executed in parallel. Each monitor registered at the bus and continued with sending a given amount of message bundles. All measurements are performed on an IBM Lenovo T60 with an Intel Core2 Duo T7200 2.0GHz, 2GB main memory and a solid state harddisk in SATA AHCI mode for booting the virtualization environment and OS. The result of the performance evaluation is shown in figure 5.

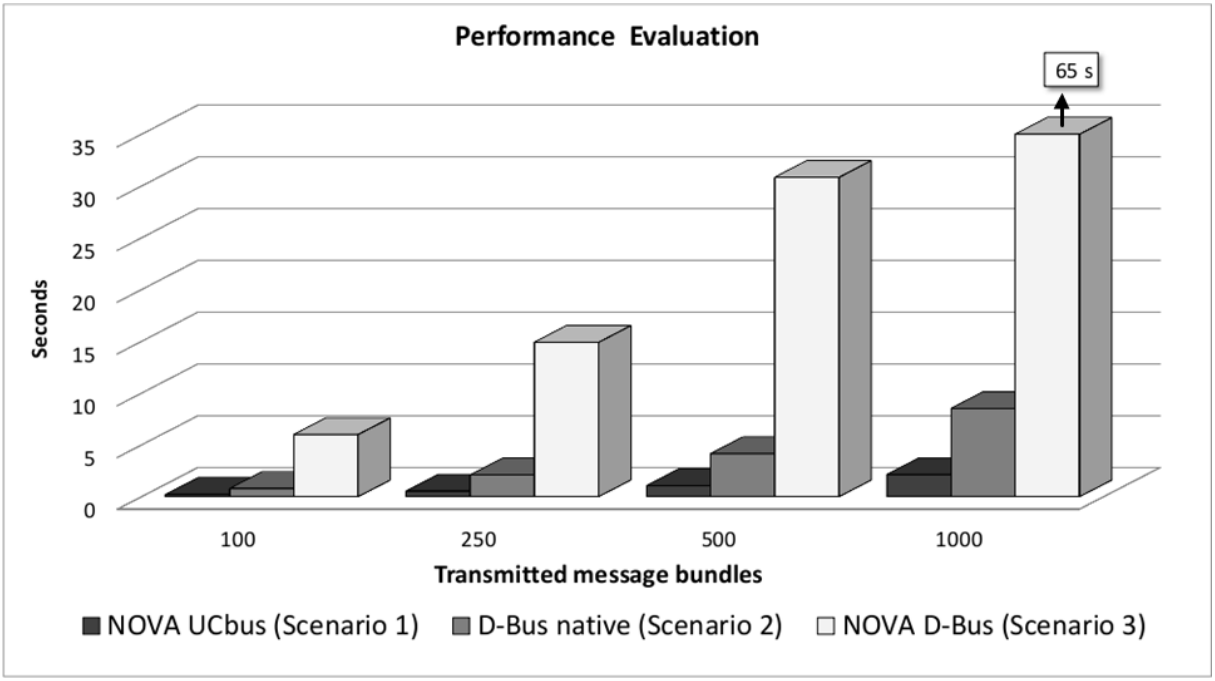

Figure 5: Performance Evaluation

We can see an almost linear correlation between time consumption and the amount of exchanged message bundles for all considered scenarios. It is evident that scenario 1, with the solution we describe and advocate in this paper, provides the fastest message exchange, followed by the D-Bus implementation in a natively booted OS. Hence, in terms of performance, our hypervisor-based solution is the fastest. By design, this includes the requirement of using the NOVA virtualization environment, which in fact decreases the overall performance of the data consumer's system. The impact of the virtualization environment itself is evaluated as comparison between scenarios 2 and 3. For this work the employment of a virtualization technique was intended, and this observation hence refers not to a drawback but rather a given fact.

In sum, our communication infrastructure provides satisfactory performance results, thus complying with requirement 3, "Minimal performance overhead." Although direct communication using well-

known inter-process communication between the monitors is faster, it implies major drawbacks conflicting to the introduced requirements, as explained in section 4.

7.2 Security

Considering attacks to the usage control environment, the main issue is a user trying to subvert installed usage control mechanisms, i.e., deployed policies. Usually the system is under exclusive control of the user, especially in the distributed environment considered in this work. Therefore, the basic attack scenario is as follows: a user in the role of data consumer inside the virtualized OS tries to circumvent the usage control environment to gain more privileges and to initiate actions the policy prohibits. This is possible using two different attacks:

1. Modifying content of messages exchanged between monitors
2. Preventing message exchange by attacking the availability of the communication infrastructure

This work's focus is on the communication infrastructure. Therefore, exploiting vulnerabilities of the monitors is out of the scope of our analysis because monitors are not part of this work and provide their own security analysis, resulting in the first assumption:

A.1 No vulnerabilities in usage control monitors

Additionally, attacks on the virtualization environment NOVA are neglected, because it is an ongoing development project currently still in a pre-release state. Thus breaking out of the VM and directly attacking the bus component might be possible, but is not further investigated in this work, leading to the second assumption:

A.2 No vulnerabilities in the virtualization environment

One important attack possibility concerns the authenticity of the monitors trying to register with the bus as introduced in section 5. A malicious user could implement his own application with the intention to store faulty or compensating policies or sending illegal notifications. Therefore, checking the authenticity of validated and trustworthy monitors is essential. Using techniques from trusted computing, also introduced later as countermeasures against other attacks, the integrity and authenticity of applications can be guaranteed. As this attack is not related to the communication infrastructure, it is mentioned only for completeness but induces a new assumption:

A.3 Monitors are authentic

The first attack scenario, modifying inter-monitor messages, requires an attacker to intercept the messages and modify the content while the message is processed in a component involved in the communication chain. Following the mentioned assumptions, possible attack points are either the userspace library or the kernel module, as shown in figure 6. The userspace library is a shared library exclusively designed to increase the usability of the communication interface for monitor developers. As all attacks are based on the dynamic linking property of the OS for the library, a possible solution for preventing such attacks is to use a static binding or directly communicating with the kernel module. The latter opportunity decreases the usability for monitor developers, but completely prevents this attack possibility.

The second potential attack point is the kernel module, mainly responsible for address translation from the user address space into guest physical addresses. The communication interface between kernel and user space using procFS depends on persistent structures for storing data related to the appropriate entry in the process filesystem. Due to the persistence during module lifetime, it can be modified by other

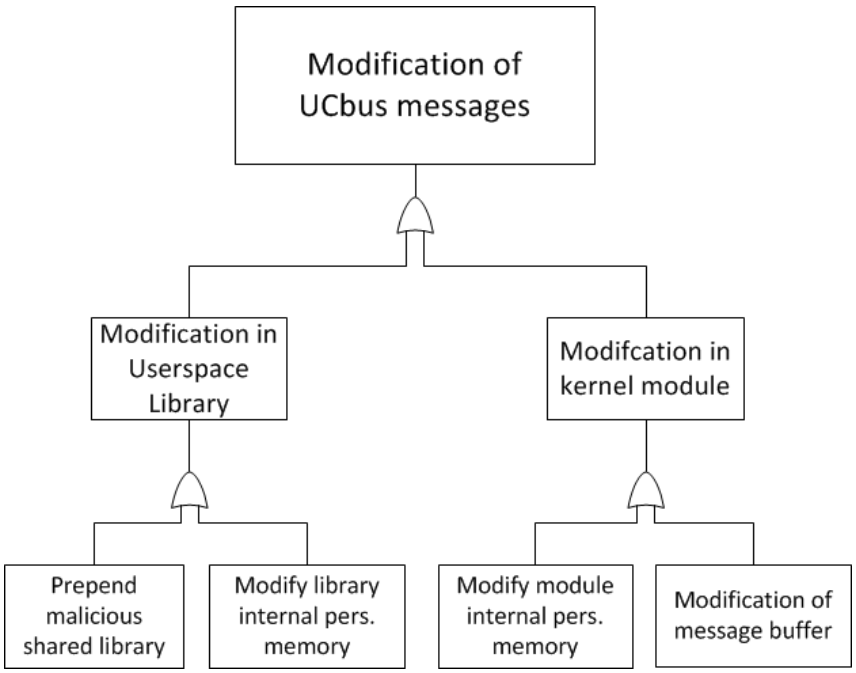

Figure 6: Attack tree: Modifying messages exchanged between monitors

kernel modules inserted by a malicious user at runtime. This modification includes the procFS entry itself (e.g. callback function pointers) and the associated internal buffer. A malicious user can change the function pointers or the buffer content to modify the notification message or the policy a monitor has requested. Both attacks are highly severe for the usage control environment, but in practice induce a race condition: the buffer content is only relevant until the receiver monitor have read it. Usually, this time slice appears sufficiently small to prevent a systematic modification of the buffer.

Considering function pointers, a kernel module is vulnerable to other potentially malicious modules. Therefore attacks to the kernel module cannot be prevented completely from the communication infrastructure itself, but rather require a protection of the kernel's integrity, provided for example by [21]. We have to assume the security of the kernel by either such a protection mechanism or disabling support for dynamically loading kernel modules and statically integrating the contributed kernel module for the communication infrastructure:

A.4 Persistent memory protection in kernel modules

The second attack scenario pertains to availability. As shown in figure 7, one approach attacks the virtualization environment. Following assumption A.1, the only way is to boot the data consumer's OS natively without the NOVA architecture to disrupt the communication chain. The presence of NOVA in the boot process can be verified by using a specific boot loader such as Trusted Grub, which creates hash values of every component involved in the boot process, e.g. boot modules, to establish a core root of trust for measurement [22]. Except the hardware requirement of a TPM, this solution introduces no further dependencies, but guarantees a valid boot sequence.

As shown by the attack scenario for modifying messages, other attack points refer to the userspace library and the kernel module. The already introduced countermeasures with using a static library or directly communicating with the kernel decrease the maintainability of the infrastructure but completely prevent this attack possibility.

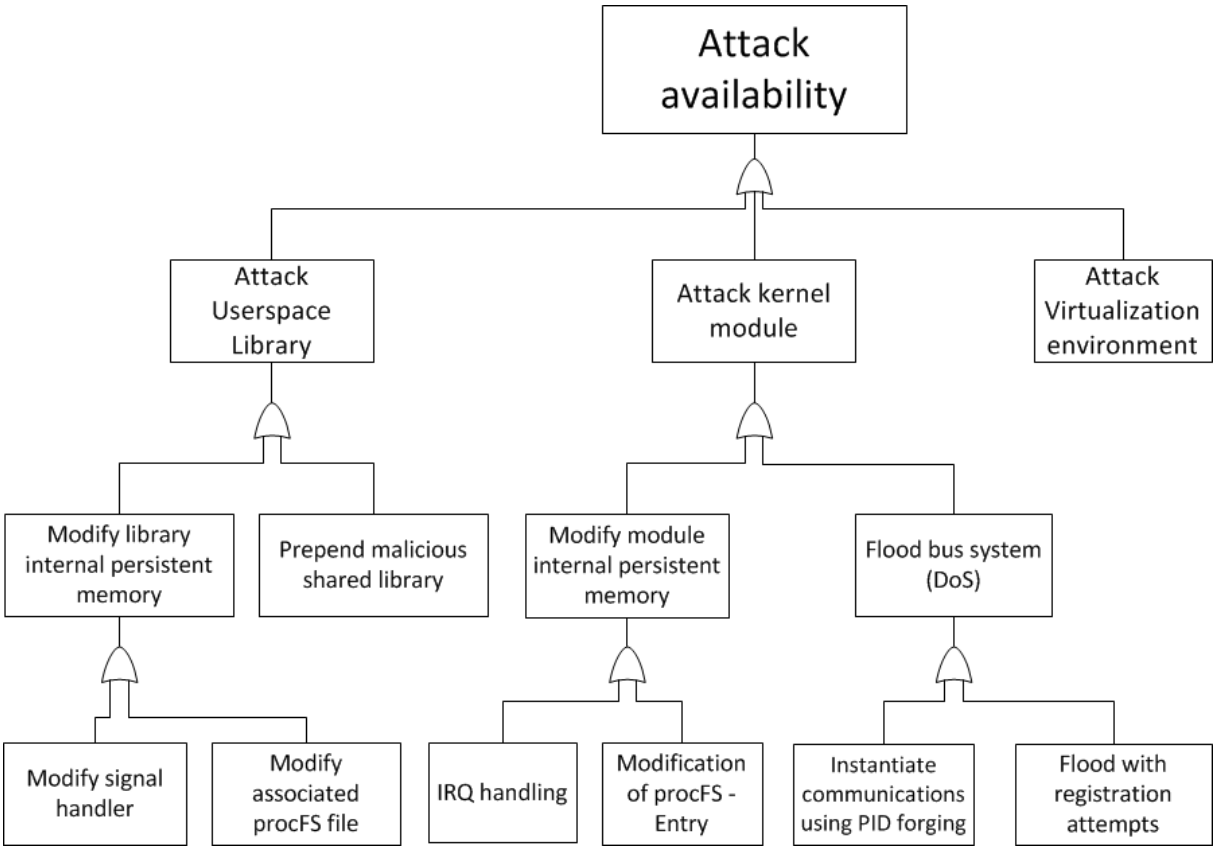

Figure 7: Attack tree: Availability of communication infrastructure

For the kernel module a new attack node is shown in the attack tree. By modifying the associated IRQ, completely deallocating the binding to the IRQ or more sophisticated attacks such as directly accessing the APIC (Advanced Programmable Interrupt Controller) or modifications at the resource allocation of the PCI structures, any notification from the UCbus remains unnoticed in the kernel module and therefore in the usage control monitors. Although this issue affects mainly notification messages between monitors, the other messages types are vulnerable to modifications of the procFS entry as well. As explained above for attacks on the kernel module for modifying the message content, either a runtime check of the kernel's integrity is required, or a trustworthy kernel without module support is used for booting the system, additionally with guaranteeing the trustworthiness of components involved in the boot procedure using Trusted Grub.

The last attack scenario on the kernel is a denial-of-service attack on the communication infrastructure. The entire communication is limited to registered monitors by checking their process identifier (PID) used in the registration. Therefore either forging the PID is necessary or a flooding with registration attempts. PID forging clearly requires a modification of the appropriate kernel mechanisms for assigning these identifiers. Although using another process ID will introduce serious problems with respect to the system's stability, this attack might be possible. The introduced checking of the kernel integrity and relying on a trustworthy kernel using Trusted Grub inhibits any malicious modification at the kernel code, thus also a forging of the PID required in this attack scenario. Another DoS uses registration messages for flooding the bus system. However trying to decrease the availability of the communication infrastructure directly falls back to the performance of the overall system of the data consumer. Therefore a malicious user basically thwarts himself with attacking the bus system using a

DoS attack. Finally this attack scenario implies a high severity but only a very limited applicability due to the explicit consequences for the attacker.

In sum, we can conclude that the communication infrastructure is secure under the mentioned assumptions. Assumption A.1 and A.3 refer to the communication endpoints in the OS, namely the usage control monitors. This work focuses on the infrastructure for message exchange. Therefore, the monitors and the virtualization environment NOVA and especially their security are not covered in this work. Finally, only assumption A.4 that requires a protection mechanism of the kernel's integrity is rather a strong condition. Using a static kernel image without module support induces high restrictions for the affected user. Providing a runtime verification of the kernel integrity requires further modifications at the system in addition to further performance overhead. However protecting the kernel against malicious modules is suggested, although the attack risk without any guardian is acceptable due to the timing issue. Using such a verification technique in addition to Trusted Grub offers the possibility to detect at least any modification for attacking the availability, and therefore the data provider can deny to deploy his data in such a case.

8 Future work

Recalling Section 7, regardless of the specific solution adopted to provide their connection, authentication of monitors is still an open question. This issue, together with an improved model for sensitive data tracking across layers of abstraction, is the subject of current investigations: without an appropriate information flow model, the automatic generation of notification messages is not possible. Moreover, a refined model and a language are needed to specify and translate policies that simultaneously address representations at different layers [4]. These issues, albeit orthogonal to the contributed solution, are mentioned for completeness, since they are relevant for our vision of a comprehensive usage control environment.

In terms of future developments, the platform-dependent components of our infrastructure should be ported to other operating systems like Microsoft Windows or Apple Mac OS X. This requires the extension of NOVA for booting these operating systems and the existence of usage control monitors for such environments.

Next, we want to investigate the feasibility of extending the communication infrastructure to connect monitors, or in general, processes, *across* different virtual machines. Although this process introduces a plethora of new technical issues, the interface between monitors at two different layers would be conceptually the same, be the two layers in the same machine or not. Moreover, the usefulness of a fast, secure and reliable way to communicate with another machine (virtualized on the same platform) is beyond mere usage control purposes, in particular considering the increasingly prominent role of cloud architectures nowadays.

Last, but not least, we want to look into deployment strategies to easily distribute our communication infrastructure to data consumers.

9 Conclusions

With this work, we addressed the problem of connecting usage control monitors at different layers of abstraction. The simultaneous existence of multiple representations of the same data within the system (and the need of connecting all of them) requires data-flow tracking monitors to exchange information, possibly across different levels of abstraction; this is required in order to maintain a consistent state of data distribution among the system. Considering the amount and the extremely sensible nature of the messages exchanged, performance and security issues are key factors to be taken into account.

As an example, we considered a picture both as content of a webpage at the browser level and as a cache file at the operating system level: if the browser creates a cache file, it must notify the OS-level monitor that that file must, from that moment onwards, be monitored as well. A proof-of-concept implementation of this use case is presented in details in Section 6 and evaluated in Section 7.

In contrast to other solutions for inter-process communication such as D-Bus or shared memory (Sections 4, 7), our approach is to create an operating system-independent bus system at the hypervisor level. This choice was motivated by the security and performance concerns explained above (and afterwards justified by evaluation results, as shown in Section 7). While analyzing existing virtualization solutions, some turned out to be inadequate for our goal, violating functional (e.g. platform independence) or non-functional (e.g. performance overhead) requirements. Other solutions, like XEN [17], or similar research projects, like BitVisor [19] or SecVisor[21], have been taken into account, but were eventually discarded when compared to the NOVA architecture (Section 2). This was due to hardware requirements, larger size of the trusted codebase or required amount of modifications to the original architecture. Among the valid open-source alternatives, in fact, NOVA seemed to be the best framework to develop our bus system (Section 2), in particular given its focus on security and its compartmentalized structure. When offered two different options for our bus integration (as a device model or as a secure application), although they were both viable alternatives, we opted for the former, due to the smaller number of steps required in the communication chain and to its easier integration into the existing architecture.

In Section 7 we evaluated our solution against various types of attacks and compared it with two alternatives. According to our experimental results, our bus system performs better than an equivalent D-Bus implementation. For obvious reasons, only attacks concerning our specific communication solution have been taken into account: vulnerabilities in external components, like NOVA or usage control monitors, and generic attacks, such as for instance, resetting the TPM component to forge a fake trusted boot [23], are out of the scope of our analysis.

Finally, despite being designed for a specific well-defined purpose in the context of cross-layer usage control enforcement, the flexibility of our bus protocol makes it suitable for many other needs requiring a fast and secure message exchange for common userspace applications. In terms of future work (Section 8), we want to investigate the issues in the monitor authentication process, the feasibility of extending our architecture to support other platforms (Microsoft Windows and Mac OS X) and the communication between processes in different virtual machines. Moreover, we work on aspects related to the deployment of our infrastructure.

Acknowledgment

This work was supported by FhG Internal Programs, Attract 692166, as well as by the Google Award CARLA.

References

[1] C. Moucha, E. Lovat, and A. Pretschner, "A hypervisor-based bus system for usage control," in *Proc. of the 6th International Conference on Availability, Reliability and Security (ARES'11), Vienna, Austria*. IEEE, August 2011, pp. 254–259.

[2] J. Park and R. Sandhu, "The UCON ABC usage control model," *ACM Transactions on Information and System Security*, vol. 7, no. 1, pp. 128–174, February 2004.

[3] A. Pretschner, M. Hilty, and D. Basin, "Distributed usage control," *Communications of the ACM*, vol. 49, no. 9, pp. 39–44, September 2006.

[4] A. Pretschner, E. Lovat, and M. Büchler, "Representation-independent data usage control," in *Proc. of the 6th international Workshop on data privacy management (DPM'11), Leuven, Belgium*, September 2011.

[5] M. Harvan and A. Pretschner, "State-based usage control enforcement with data flow tracking using system call interposition," in *Proc. of the 3rd International Conference on Network and System Security (NSS'09), Gold Coast, Queensland, Australia*. IEEE, October 2009, pp. 373–380.

[6] A. Pretschner, M. Buechler, M. Harvan, C. Schaefer, and T. Walter, "Usage control enforcement with data flow tracking for X11," in *Proc. of the 5th International Workshop on Security and Trust Management (STM'09), Saint Malo, France*, September 2009, pp. 124–137.

[7] M. Dam, B. Jacobs, A. Lundblad, and F. Piessens, "Security monitor inlining for multithreaded java," in *Proc. of the 23rd European Conference on Object-Oriented Programming (ECOOP'09), Genova, Italy, LNCS*, vol. 5653. Springer-Verlag, July 2009, pp. 546–569.

[8] I. Ion, B. Dragovic, and B. Crispo, "Extending the Java Virtual Machine to Enforce Fine-Grained Security Policies in Mobile Devices," in *Proc. of the 23rd Annual Computer Security Applications Conference (ACSAC'07), Miami Beach, Florida, USA*. IEEE, December 2007, pp. 233–242.

[9] L. Desmet, W. Joosen, F. Massacci, K. Naliuka, P. Philippaerts, F. Piessens, and D. Vanoverberghe, "The S3MS.NET Run Time Monitor: Tool Demonstration," *Electronic Notes in Theoretical Computer Science*, vol. 253, no. 5, pp. 153–159, December 2009.

[10] U. Erlingsson and F. Schneider, "SASI enforcement of security policies: A retrospective," in *Proc. of the 1999 workshop on New security paradigms (NSPW'99), Caledon Hills, Ontario, Canada*. ACM, 1999, pp. 87–95.

[11] B. Yee, D. Sehr, G. Dardyk, J. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," *Communications of the ACM*, vol. 53, no. 1, pp. 91–99, January 2010.

[12] A. Pretschner, J. Ruesch, C. Schaefer, and T. Walter, "Formal analyses of usage control policies," in *Proc. 4th International Conference on Availability, Reliability and Security (ARES'09), Fukuoka, Japan*. IEEE, March 2009, pp. 98–105.

[13] A. Pretschner, F. Schütz, C. Schaefer, and T. Walter, "Policy evolution in distributed usage control," *Electronic Notes in Theoretical Computer Science*, vol. 244, pp. 109–123, August 2009.

[14] J. S. Robin and C. E. Irvine, "Analysis of the intel pentium's ability to support a secure virtual machine monitor," in *Proc. of the 9th conference on USENIX Security Symposium - Volume 9, Denver, Colorado*. USENIX Association, 2000, pp. 10–10.

[15] "Intel software developer's manual," 2010.

[16] U. Steinberg and B. Kauer, "Nova: A microhypervisor-based secure virtualization architecture," in *Proc. of the 5th European conference on Computer systems (EuroSys '10), Paris, France*. ACM, April 2010, pp. 209–222.

[17] XenProject, "Xen cloud platform," May 2010, http://www.xen.org/products/cloudxen.html.

[18] "D-Bus," 2011. [Online]. Available: http://dbus.freedesktop.org

[19] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato, "Bitvisor: a thin hypervisor for enforcing i/o device security," in *Proc. of ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE'09), Washington, D.C., USA*. ACM, March 2009, pp. 121–130.

[20] K. Kaneda, "Tiny virtual machine monitor," 2006, http://web.yl.is.s.u-tokyo.ac.jp/~kaneda/tvmm/.

[21] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *Proc. of the 21st ACM SIGOPS symposium on Operating systems principles (SOSP'07), Stevenson, Washington, USA*. ACM, October 2007, pp. 335–350.

[22] R. Neisse, D. Holling, and A. Pretschner, "Implementing trust in cloud infrastructures," in *Proc. of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID'11), Los Alamitos, California, USA*. IEEE, May 2011.

[23] Dartmouth College PKI/Trust Lab, "TPM reset attack," Sep 2010, http://www.cs.dartmouth.edu/~pkilab/sparks/.

Cornelius Moucha received his Bachelor of Science in Network Computing 2008 from the Technical University Freiberg, Germany and afterwards the Master of Science 2011 from the Technical University Kaiserslautern, Germany. Currently he is working as an engineer in the department Information Systems Quality Assurance at the Fraunhofer Institute for Experimental Software Engineering IESE in Kaiserslautern, Germany. His research interests include information security, in particular usage control, and system security.

Enrico Lovat received a Master degree in Computer Science in 2009 from the University of Verona, Italy. He worked as researcher at Fraunhofer Institute for Experimental Software Engineering in Kaiserslautern (Germany). Since 2010 he is a Ph.D. student in Prof. Pretschner's group at Karlsruhe Institute of Technology, Germany. His research activities focus on information security, in particular usage control and information flow.

Alexander Pretschner is a full professor of computer science at Karlsruhe Institute of Technology, Germany, where he heads the Certifiable Trustworthy IT Systems group. Master's degrees from RWTH Aachen and the University of Kansas, PhD degree from TU Munich. Research interests include software engineering, specifically testing, and information security, specifically distributed data usage control.