A Hypervisor-Based Bus System for Usage Control

Cornelius Moucha
Fraunhofer IESE, Kaiserslautern, Germany

Enrico Lovat, Alexander Pretschner
Karlsruhe Institute of Technology, Germany

***Abstract*—Data usage control is concerned with requirements on data after access has been granted. In order to enforce usage control requirements, it is necessary to track the different representations that the data may take (among others, file, window content, network packet). These representations exist at different layers of abstraction. As a consequence, in order to enforce usage control requirements, multiple data flow tracking and usage control enforcement monitors must exist, one at each layer. If a new representation is created at some layer of abstraction, e.g., if a cache file is created for a picture after downloading it with a browser, then the initiating layer (in the example, the browser) must notify the layer at which the new representation is created (in the example, the operating system). We present a bus system for system-wide usage control that, for security and performance reasons, is implemented in a hypervisor. We evaluate its security and performance.**

***Keywords*-Usage Control, Virtualization, Information Flow**

I. Introduction

Usage control [1], [2] generalizes access control to what happens to data after access has been granted. This is particularly difficult in distributed settings. Among other domains, distributed usage control is relevant in the context of data protection, management of IP (e.g., in the cloud), compliance with regulations such as HIPAA or SOX, 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 that it is rendered on a screen or a loud speaker or a piece of paper; if the data is executable, it may be executed; it can be modified with, among others, a word processor or a graphics tool; it can also be copied and disseminated or deleted. If such a usage takes place, the data usually is transformed into an additional representation. For instance, a picture can exist as window content, as a file, as a pixmap in X11, as a Java object, and as part of a text document. If usage control requirements are to be enforced on *data*, then this likely means that they have to be enforced on *all representations of the data*. As a consequence, data flow tracking systems have been built recently that track the flow of data within one layer of abstraction (e.g., [3]). The idea essentially is to identify relevant actions at each layer and give them a semantics that captures the flow in-between different representations. At runtime, the mapping from data to representations is updated upon execution of a relevant action. In this way, for instance, we can track the flow of data in-between memory regions, files, and network sockets at the layer of the operating system (OS, [3]).

It might appear simpler to not assume one data flow tracking (and hence usage control enforcement) engine per layer but rather exclusively perform the tracking at the level of machine code. However, requirements such as "don't copy" have different semantics at different layers of abstraction (copy&paste in a text editor, copy a file, clone an object), and it is almost impossible to automatically identify those parts of the machine code that pertain to these high-level actions. For this reason, the layered approach appears like a viable alternative.

Evidently, it is not sufficient to perform data flow tracking (and related, usage control enforcement) at each layer in isolation. Consider a browser that downloads a picture from a network. Once it has downloaded the picture, the browser renders it and creates a cache file. Hence, there are at least three different representations: the pixmap on the screen, the cache file and internal representations in the browser. If the browser receives the picture together with a policy then the policy must be communicated from the application layer (the browser) at least to the layers of the operating and the windowing systems. Moreover, 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.

With this work we tackle the problem of how to communicate (1) policies and (2) the flow of data from one layer of abstraction to another layer. Our solution consists of a bus system that we implemented at the hypervisor layer, both for security and performance reasons that we evaluate.

Contribution: As far as we are aware, nobody has considered representation-independent usage control before. As a consequence, our contribution consists 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 the problem of the enforcement at single layers of abstraction, conflicting policies, and not the problem of delegation.

II. Background: Virtualization

Virtualization is a method of logically dividing computer resources of a physical machine, mainly hardware such as processor, main memory, network connectivity and others, into separate virtual machines executing their instructions

independent of each other. The virtual machine monitor (VMM) or *hypervisor*, is an application responsible for providing an environment in which several virtual machines share the resources of one single host. The VMM and hypervisor can also be separated to distinguish the managing entity of the guests and of the host. Hypervisors are divided in two categories, differing in the capability of directly accessing hardware [4]: A *type-1 hypervisor* is a specialized system running directly on the host's hardware and managing access requests from guest OSs. 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 doesn't perform as efficiently as the type-1 hypervisor but, on the other hand, does not pose any special constraint on the underlying hardware platform.

As explained in Section III, our requirements for the bus infrastructure include the possibility of running the communication infrastructure for a generic preinstalled OS and with a minimum overhead. This precludes the use of a type-2 hypervisor, due to the decreased performance generated by the additional OS, and requires the adoption of a full virtualization solution, considering the guest kernels' modifications needed by paravirtualization (type-1 hypervisor solutions).

Recent processor generations provide hardware support for virtualization directly integrated in the CPU, namely Intel-VT or AMD-V. In addition to facilitate two different operation modes, for virtual machines and hypervisor respectively, these virtualization extensions offer an additional instruction, "VMCALL" [5], for switching the execution context from the guest OS (VMX non-root) to the associated virtual machine monitor (VMX root). This particular instruction will be the key support for the communication between bus infrastructure (host) and monitors (guest).

One virtualization environment for the desired communication infrastructure is the NOVA OS Virtualization Architecture [6], a type-1 hypervisor solution. 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. NOVA provides a suitable framework for our work, mainly due to its architectural design and in particular to the component-based development's 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.

III. Requirements

Our goal is to provide a secure and fast communication infrastructure for usage control-related message exchange across different levels of abstraction. The infrastructure should also provide an interface for storing and retrieving data, in this case policies and usage information, from/to a predefined secure location. 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 inside an identical 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 explained before, the bus system has to be accessible independently of the data consumer's OS in which usage control monitors are executed. The solution should produce as little performance overhead as possible. Security has to be considered as well, including tamper-resistant message exchange, secure storage location for policies and constant availability of the communication system. Data consumers usually have (OS-)administrator privileges and therefore can easily terminate running applications and services, including existing bus systems. According to the security requirements this must not be possible in our solution.

We hence identify the following requirements: (1) Infrastructure for information exchange: (1.1) Platform independence of bus implementation, (1.2) Minimal modifications to existing OS and virtualization architecture, (1.3) Virtualize one preinstalled OS and extend it to use bus functionalities; (2) Shared, secure location for 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. As far as we know, bus systems for standard applications out of the OS's scope currently do not exist.

IV. Design

The NOVA architecture (Section II) 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. Another option is to implement the bus component as a virtual device model, a purely software-based abstraction of hardware used by the guest system to access 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 with the guest VM and can also be designed to access the bus component, without providing any device-specific functionality to the guest OS.

Although only one guest OS is intended to be used in our context, separating the bus system from the actual 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.

Sticking to the single-guest scenario addressed by our work, we decided to use the virtual device model. 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 message scheme design 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. By initiating the VMM, designated virtual devices are instantiated. Both communication partners, the VMM and the bus component, can exchange messages using already existing communication interfaces for device models. This minimizes the necessary 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: Once a communication is initiated by a usage control monitor, the first involved entity is the OS kernel. This component is responsible for translating the given memory address 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, need to read the message (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 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 address space of the VM itself. The partition manager provides an interface for this translation. Finally the last communication step is under the responsibility of the VMM to forward the message with the translated memory address to the bus component.

Communication protocol: In order to exchange usage control information, different types of messages have to be considered. 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. In order to identify valid communication partners, a process identifier (PID) must also be submitted during the registration process (see Section VI, assumption A2). Additionally, registration allows for 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. As the communication back to the guest system operates asynchronously via interrupts, the notification 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.

V. Implementation

The userspace library, the connector to the monitors, is designed as a common API library so that it can be used by monitor developers. It has to provide an interface for encapsulating the required communication with the kernel module for all introduced methods from 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 communicating with user applications. To separate the monitor registration from other message types, one communication line is solely dedicated to registrations whereas another 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.

VI. Evaluation

Requirement 1 (Section III) and its sub-requirements are met by design.

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 (Scenario 1)

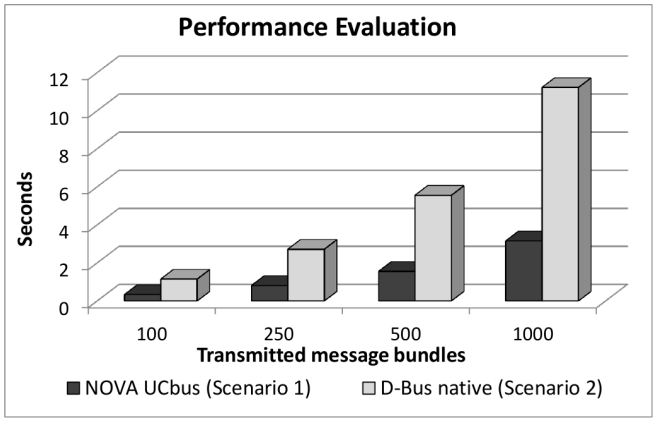

Figure 1. Performance Evaluation

to a native D-Bus-based [7] implementation (Scenario 2) providing the same functionality and message interface.

For performance evaluation, we considered messages of different types because they require different processing by the bus component: messages related to policy storage require a higher computational effort whereas for notifications the communication chain is more complex. In realistic application scenarios, monitors exchange several messages of different types, therefore it is reasonable to evaluate bundles of heterogeneous messages combined. These bundles consist of one message for storing and for retrieving a policy and one notification message. Policy retrieval as well as notification messages use random values as destination monitor and for the concerned policy.

For both 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 for booting the virtualization environment and OS. The result of the performance evaluation is shown in Fig. 1.

We can see an almost linear correlation between time consumption and the amount of exchanged message bundles for the considered scenarios (1 and 2). It is evident that in this context the solution we describe and advocate in this paper performs better, almost 5 times faster than an equivalent D-Bus implementation in a natively booted OS. The requirement of using the NOVA virtualization environment introduces the drawback of decreasing the data consumer system's overall performance of a negligible 1-3% [6].

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 with the introduced requirements.

Security: Considering attacks to the usage control environment, the main issue is a user trying to subvert usage control mechanisms. Therefore, the basic attack scenario is a user that, as data consumer, tries to circumvent usage control restrictions in the virtualized OS to gain more privileges and to initiate actions otherwise prohibited by the policy. This is possible using two different attacks: (1) modifying message content exchanged between monitors, or (2) preventing message exchange by attacking the availability of the communication infrastructure.

This work's focus is on the communication infrastructure. Exploiting vulnerabilities of the monitors is out of the scope of our analysis because monitors are not part of this work. Therefore it's reasonable to assume **[A.1]** *no vulnerabilities in usage control monitors.*

In addition, monitors must also be authentic: a malicious user does not need to get control of existing monitors if he can implement his own, register it to the bus and store faulty policies or send illegal notifications. The authenticity of applications can be guaranteed by using techniques from trusted computing. This attack not strictly being related to the communication infrastructure, we just mention it for completeness and take a new assumption: **[A.2]** *monitors are authentic*.

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 third assumption: **[A.3]** *no vulnerabilities in the virtualization environment*.

Modifying inter-monitor messages: The first attack scenario requires an attacker to intercept the messages or modify the content while the message is processed in a component involved in the communication chain. Under the aforementioned assumptions, possible attack points are either the userspace library or the kernel module. 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, a possible solution for preventing such attacks is to use a static library 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 responsible for address translation from the user address space into guest physical addresses. The communication interface between kernel and user space (procFS) depends on persistent structures for storing data related to the appropriate entry in the process file. Due to the persistence during

module lifetime, it can be modified by other 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 reads it. Normally, 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 [8]. We have to assume the security of the kernel by either such a protection mechanism or by statically integrating our module for the communication infrastructure and disabling support for loading other modules: **[A.4]** *persistent memory protection in kernel modules*.

Denial of service: The second attack scenario concerns availability. As previously mentioned, one approach would be to attack the virtualization environment. Under assumption A.1, the only way to do it is to boot the data consumer's OS natively, without the NOVA architecture, in order to disrupt the communication chain. The presence of NOVA in the boot process can be verified by using a specific boot loader like Trusted Grub, which creates hash values of every component involved in the boot process, like boot modules, to establish a core root of trust for measurement [9]. Except for the hardware requirement for a TPM, this solution introduces no further dependencies.

Other entry points for an attack are the userspace library and the kernel module. The countermeasures introduced above of using a static library or directly communicating with the kernel decrease the maintainability of the infrastructure but completely prevent this attack possibility.

Furthermore, by modifying the associated IRQ or completely deallocating the binding to the IRQ in the kernel, any notification from the UCbus would remain unnoticed by the kernel module, and therefore invisible to any usage control monitor. Although this issue affects solely notification messages, the other messages types are vulnerable to modifications of the procFS entry as well. Once again, either a runtime check of the kernel's integrity or booting the system with a trustworthy kernel without module support (e.g., via Trusted Grub) would make the attack unfeasible.

The last attack scenario concerning the kernel is a denial-of-service attack on the communication infrastructure. By design, communication is restricted to registered monitors by checking their process identifier. Therefore only forging a PID or flooding with registration attempts is possible. PID forging clearly requires a modification of the kernel mechanisms for assigning these identifiers. Although faking another process ID will introduce serious system's stability issues, it might be possible. Checking the kernel's integrity inhibits any malicious modification at the kernel code, including forging the PID, fundamental in this attack.

Alternatively, DoS may be achieved by flooding the bus with registration messages and can't be prevented. However, decreasing the availability of the communication infrastructure directly falls back on 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 four assumptions: assumption A.1 and A.2 refer to the communication endpoints in the OS, namely the usage control monitors, out of the scope of this work; similarly, the virtualization environment NOVA and especially its security are not covered in this work, inducing assumption A.3. The only rather strong condition is then assumption A.4, that requires a protection mechanism of the kernel's integrity. 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 [9]. However protecting the kernel against malicious modules is suggested, although the attack risk without any guardian is acceptable, due to the race condition timing issues. Using such a verification in addition to Trusted Grub offers the possibility to detect any modification for attacking the availability, and therefore the data provider can deny to deploy his data.

VII. Related Work

Related work has been cited throughout the text. In addition, most virtualization environments provide the possibility for inter-domain communication between virtual machines. The Xen hypervisor uses XenBus for this connection [10]. Furthermore, several communication interfaces inside the OS exists. This includes common bus systems like D-Bus [7] and general inter-process communication like local sockets or shared memory. Several existing research projects introduce hypervisors dedicated to different specialized functionalities, like Bitvisor [11], Tiny Virtual Machine Monitor [12] or SecVisor [8], a hypervisor dedicated to security functionality, especially harddisk encryption. Although all of the presented possibilities can be extended for solving our mentioned 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.

VIII. Conclusions

We addressed the problem of connecting usage control monitors at different levels of abstraction. This connection is needed to relate different representations of the same data within the system. 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 this file must, from this moment onwards, be monitored as well. Regardless of the concrete data-flow model adopted, only a synergy of the layer-specific monitors can maintain a coherent and reliable trace of data distribution among the system; thus the monitors need to "talk" to each other. Considering the amount and the extremely sensible nature of the messages they exchange, performance and security issues must be taken into account.

Instead of a standard solution for inter-process communication or bus system such as D-Bus (Section VI), we decided to take a different approach: virtualize the whole system and implement the communication infrastructure 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 VI). Several virtualization solutions turned out to be inadequate for our goal, due to the infringement of functional or non-functional (platform independence, performance overhead) requirements: among the valid open-source alternatives, we finally decided to integrate the bus system into the NOVA architecture (section II), because of its focus on security and its compartmentalized structure. Due to hardware requirements, a larger size of the trusted codebase or required amount of modifications to the original architecture, other familiar solutions, like XEN [10], or similar research projects, like BitVisor [11] or SecVisor[8], have been discarded.

Our solution has been showed to be secure against various types of attacks and to perform better than an equivalent D-Bus implementation (Section VI). For obvious reasons, only attacks concerning our specific solution have been taken into account: vulnerabilities in external components, like NOVA or usage control monitors, and generic attacks, like, for instance, reset the TPM component to forge a fake trusted boot [13], are out of the scope of our analysis.

In terms of future work, authentication of monitors is still an open question, independent of the solution that provides their connection. This issue, together with an information flow model for sensitive data tracking across several layers of abstraction, will be subject of further investigation. Basically, an adequate information flow model is required for an automatic generation of notification messages. Furthermore, an abstract policy specification is necessary to address more than one layer of abstraction at the same time. These issues for further research are independent of the contributed solution, but are mandatory for providing an overall usage control environment and thus mentioned for completeness.

Finally, considering the increasingly prominent role of cloud and virtualization architectures, we want to investigate the feasibility of extending the communication infrastructure to connect monitors, or in general, processes, across different virtual machines. 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. Last, but not least, deployment strategies for the contributed solution should be further considered to provide an easy and flexible procedure for distributing the communication infrastructure to data consumers.

Acknowledgment

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

References

[1] J. Park and R. Sandhu, "The UCON ABC usage control model," *ACM Trans. Inf. Syst. Secur.*, pp. 128–174, 2004.

[2] A. Pretschner, M. Hilty, and D. Basin, "Distributed usage control," *Commun. ACM*, vol. 49, pp. 39–44, September 2006.

[3] M. Harvan and A. Pretschner, "State-based usage control enforcement with data flow tracking using system call interposition," in *Proc. NSS*, 2009, pp. 373 –380.

[4] J. S. Robin and C. E. Irvine, "Analysis of the intel pentium's ability to support a secure virtual machine monitor," in *Proc. 9th USENIX Security Symposium*, 2000, p. 10.

[5] "Intel software developers manual," 2010.

[6] U. Steinberg and B. Kauer, "Nova: A microhypervisor-based secure virtualization architecture," in *Proc. EuroSys*, Apr 2010, pp. 209–222.

[7] "D-bus." [Online]. Available: http://dbus.freedesktop.org

[8] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *Proc. 21st ACM SIGOPS symp. on Operating systems principles*, 2007, pp. 335–350.

[9] R. Neisse, D. Holling, and A. Pretschner, "Implementing trust in cloud infrastructures," *11th IEEE International Symposium on Cluster Computing and the Grid*, 2011.

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

[11] 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*, 2009, pp. 121–130.

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

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