In Guards we trust: Security and Privacy in Operating Systems revisited

Michael Hanspach
Fraunhofer FKIE, Wachtberg, Germany
Email: michael.hanspach@fkie.fraunhofer.de

Jörg Keller
FernUniversität in Hagen, Germany
Email: joerg.keller@fernuni-hagen.de

***Abstract*—With the rise of formally verified micro kernels, we finally have a trusted platform for secure IPC and rigorous enforcement of our mandatory access control policy. But, not every problem in computer security and privacy could possibly be solved by a trusted micro kernel, because we have higher level security and privacy concepts like packet filtering, data encryption and partitioning of shared hardware devices, which we also need to trust. Numerous authors have described the need for a trusted middleware, fulfilling these higher level security and privacy goals, but detailed requirements for the different security and privacy goals are still missing. We provide a collection of output filters that can be applied to trusted operating system components to enforce higher level security goals. We further provide a typology of operating system guards, which are essentially trusted components utilizing different compilations of input and output filters. The storage guard, the audio filtering guard and the sequencing guard are specifically targeted at providing solutions to three common security and privacy problems in component-based operating systems. Finally, we develop a guard reference architecture and present the concept of a guard construction kit for the development of new types of operating system guards, enabling operating system developers to build their own guard components for both component-based and commodity operating systems.**

I. Introduction

With the advent of formally verified micro kernels, trustworthy computing systems are becoming more and more a viable option. However, as micro kernels provide only basic functionalities like secure IPC (inter process communication and inter partition communication in component-based operating systems) to operating systems, higher level security critical operating system services are not targeted at. Examples for these security critical services would be encryption (on disk or over the network), packet filtering, partitioning of untrusted hardware and content filtering. Any of these services cannot be possibly integrated into a micro kernel without giving up the paradigm of small code size. Still, these services need to be implemented as trusted components, because, if we cannot trust our higher level services, the operating system's application partitions cannot build up trust in these critical tasks.

We argue that formally verified micro kernels, irrespective of all their merits, only solve the security problem at a specific operating system layer and delegate the rest of the problem to a higher operating system layer. At this higher layer, a service middleware between the kernel and our applications needs to be established. Different concepts for such a secure middleware exist (e.g. the MILS architecture), but they mostly focus on the general system architecture and not on solutions for solving our specific higher level security goals such as partitioning of shared resources or applications of input and output filtering. Some of the middleware's components would have to be implemented as a trusted component, whereby others could be implemented as untrusted components, because their critical aspects could as well be checked or provided by a trusted component. Trusted components that provide critical services to other components are known as *guards* (see also Section II). They are utilized in the context of a component-based operating system, i.e. an operating system that consists of a small, verifiable kernel and supporting components, as described by Jaeger et al. [1]. For protecting other operating system components and application partitions (see also Section III), guards utilize different input and output filters, transforming data in a way that prevents information leaks.

In this article, we present a typology of these operating system guards and a collection of their output filtering mechanisms. We further provide formal semantics for the description of operating system guards and filters, enabling operating system developers to propose and describe new types of guards. Output filter types are mapped to operating system guard types and specific guard types are described in more detail. The guard reference architecture is introduced to describe a general design pattern for the development of operating system guards. Before we present our formal semantics for guards and filters, we discuss preliminary studies in the following section.

II. Related Work

Heiser et al. [2] play an intellectual game on the question, what would happen if you could actually trust your kernel. We provide our own answer to the question, stating that many problems in today's operating systems actually would not dissolve, unless we also concentrate on providing a trusted middleware specifically targeting the security goals considered in the operating system's design. Klein et al. [3], [4], [5], Andronick et al. [6], [7] and Heitmeyer et al. [8] present formal verification approaches on the kernel level, while Chong et al. [9] developed formal semantics for the specification of a component-based operating system. We argue that the application of formal methods to micro kernels and component-based operating systems is a big step forward to provable security, but still, the specific filtering problems in trusted middleware have to be solved. This does not only involve formal verification of the trusted middleware, but especially the concepts for component interconnection, separation and filtering in information

978-0-7695-5137-1/13 $26.00 © 2013 IEEE
DOI 10.1109/SocialCom.2013.87

flows, which are discussed and extended in this article.

Anderson [10] introduces the concept of a reference monitor, governing all information flows in the operating system. While a reference monitor is fundamental to any trustworthy computing system, governing the information flows between the components is not enough, as we have to make sure that security critical components actually do the right thing (and do it correctly). Rushby [11] introduces the separation kernel, which simulates a distributed environment inside the operating system, providing isolated and strict verification of information flows. McDermott et al. [12] introduce separation VMMs, which are based on the concept of strict partitioning of operating system components, but, unlike the separation kernel, utilize fewer communication paths and do not rely on full-fledged formal verification. In a thematically related statement Neumann [13] argues that concepts like the separation kernel, published as early as 1981, have not been widely adopted in the past as these revolutionary approaches have too many preconditions for being successful and markets are adapting new technologies primarily for short-term earnings rather than rewarding foresight to radical approaches. Separation VMMs could, therefore, be both an evolutionary step to full-fledged trustworthy computing systems and an alternative approach to component-based operating systems, where lower security standards would be acceptable in exchange for simplified system development and integration. With separation VMMs, component-based operating systems are more likely to enter the broad commercial market, only utilizing commodity hardware and commodity guest operating systems in the application partitions.

Hofmann et al. [14] try to cross a bridge between commodity operating systems and formally verified systems by suggesting a hypervisor that verifies the behavior of a commodity operating system in a scenario of specific high-assurance applications running alongside untrusted applications. In contrast to evolutionary approaches, Solworth [15] argues for discontinuity in trustworthy computing systems, giving up compatibility with current concepts and architectures in favor for security-focused paradigms that have been researched for a long time but have only scarcely been applied. We argue that filters and guards, as presented in this article, will become more and more relevant to the broader market when concepts like separation VMMs are getting established in business machines and computing systems for the end-user, while the general concepts we developed apply to both separation kernels and separation VMMs, making them adaptable to different approaches to trustworthy computing and independent from a possible paradigm shift.

Bellovin [16] and Bratus et al. [17] argue that virtualization, while often providing advanced isolation measures, does not really address the security problems of IPC between cooperating applications. We aim to address IPC by development of guards and filters to prevent security and privacy leaks in VM-based computing systems and even in distributed and very communication-oriented applications. Jaeger et al. [1] present a security architecture for component-based operating systems. The authors also discuss access privilege delegation in component-based operating systems, which we address by means of the policy guard.

Alves-Foss et al. [18], Jacob [19] and many more authors argue for the adoption of the MILS architecture (based on a separation kernel) for high-assurance scenarios. Alves-Foss et al. argue that "the middleware layer is responsible for filtering out any messages that are not appropriately labeled before delivering them to the recipient". We will further discuss these message filtering mechanisms in the next section where we introduce different filter types, which will be applied to different operating system guards in the course of this article. The growing potential for application of trusted middleware is discussed by different authors. While Camek et al. [20] mention the necessity for implementing a distributed MILS architecture in ICT of future cars, Partridge et al. [21] present a secure network in space utilizing a MILS architecture. Moreover, Li et al. [22] suggest the adoption of a security middleware for software defined radio devices. Different system architectures for multilevel security are further analyzed by Levin et al. [23].

Karger [24] presents multilevel security requirements for hypervisors, explicitly differentiating between *pure isolation* and *sharing hypervisors*. We pick up these terms to describe the different kind of problems addressed by our concepts for guards. While most of these guards are designed for isolation purposes, the concept of a downgrading guard addresses the sharing hypervisor, preparing data for downgrading and sharing from higher to lower classified partitions. The concept of a sharing guard instead addresses communication from lower to higher classified partitions (i.e. with the Bell-LaPadula model [25] applied). Lampson [26] introduces *covert channels* and *storage channels*, which are specifically addressed by our guards. Storage channels are targeted by the storage guard and covert channels are, for instance, targeted by the audio filtering guard. Zhou and Alves-Foss [27] discuss refinement patterns for component-based operating system architectures like decomposition, aggregation and elimination of components. We will pick up these terms to describe different variants of operating system guards.

Robinson and Alves-Foss [28] present the MLS file server which is essentially a locally and remotely distributed multilevel file system, offering high level file system commands like *read*, *create* and *delete*. In a related approach, Robinson et al. [29] present a middleware for label-based, classification-aware filtering, the *GIOP guard*. While the MLS file server approach cannot separate between different classification levels (e.g. secret and top secret classified data) in one data partition, the GIOP guard makes a distinction between the corresponding read, write and create operations (e.g. secret-read and top-secret-read), allowing for a distinct handling of these operations on a single data partition. Both MLS file server and GIOP guard make use of an MMR (MILS message router), which acts as a decision handler for IPC between the different involved operating system partitions. While the MLS file server in conjunction with the GIOP guard could be conceived as a storage guard approach for use in a sharing hypervisor, the MILS message router acts in a similar fashion as the policy guard, introduced in a later section of this article.

Robinson et al. [28], [29] and Alves-Foss et al. [18], however, do not address the problem of malware residing in the untrusted parts of the operating system (such as the storage device driver), enabling an attacker to maliciously send data from the wrong storage section to the storage

guard. Our approach to implementing a storage guard does not feature information exchange between different partitions, which can be addressed by a different operating system guard. We, instead, implement confidentiality and integrity checks in our storage guard approach, targeted at preventing any storage channel between two partitions p_i and p_j in the presence of untrusted components handling the actual storage access.

Alves-Foss et al. [18] define a guard as a "trustworthy application which is responsible for analyzing the content of the communication and determining whether this communication is in accordance with the system security policy. The guard has the ability to modify the contents of the message, delete the message or send a constructed response back". Building upon this notion of a *guard*, we will present a collection of different guard types, utilizing very different types of filters. Having discussed the work of other authors, we will now provide our own formal definition of an operating system guard.

III. Definition of an operating system related Guard

A component-based operating system has been described by Jaeger et al. [1] as an operating system that has "a small, fixed, trusted computing base (TCB) and composes its system services from individual components". We take a look at the operating system stack of a component-based operating system (Fig. 1).

Application Partitions (P)
Middleware (M)
Kernel (K)
Hardware (H)

Fig. 1. Operating system stack of component-based operating systems

On top of the underlying hardware H and the operating system kernel K, we distinguish between application partitions and middleware partitions, which provide services to the application partitions. Partitions are strictly isolated containers inside the operating system that can only communicate with each other over K and in line with the system's security policy.

We define the set of application partitions as $P := \{p_0, \cdots, p_{n-1}\}$. The tuple of middleware partitions is defined as $M := (G, C)$ where G represents the partitions of trusted components that act as a guard defined by $G := \{g_0, \cdots, g_{k-1}\}$ and C represents the partitions of non-guard-components defined by $C := \{c_0, \cdots, c_{m-1}\}$.

As guards belong to the trusted components in a component-based operating system, they have to be subject to rigorous evaluation (and possibly formal verification). In order to be reasonably evaluatable, trusted components also have to be implemented with a very low number of source lines of code. So, in terms of trust, similar conditions as those imposed on micro kernels apply. In contrast to the trusted components, untrusted components often consist of legacy software like specific device drivers that are reutilized for compatibility reasons and for the purpose of faster system development. Each guard can utilize numerous different input and output filters to solve a specific security or privacy goal in IPC. A filtering function $\phi(d)$ is applied on data transmissions d in a way that prevents leakage of critical data. It should be noted that a filter contains more than just the filtering function of the data as it might involve side effects on the guard, for instance, stateful packet inspection in packet filters or stateful storage of hash sums for the purpose of future integrity evaluation. The relation between guards and filters can be described as a many-to-many relation, as each guard can utilize different filters and each filter can be applied by different guards. We specifically define an operating system guard g as the 5-tuple $g := (IF_{in}, IF_{out}, F_{in}, F_{out}, s)$ where IF_{in} and IF_{out} are defined as sets of input and output interfaces to other operating system components available to g. F_{in} and F_{out} are defined as sets of input and output filters, whereby a single filter f is defined by the 3-tuple $f := (\tau, if*, \phi)$. τ represents the applied filter type, describing the procedures for updating the guard's state and calling the filtering function ϕ, and $if*$ is a reference to if, defined by $if \in IF := IF_{in} \cup IF_{out}$, representing the interface where this filter is applied. Finally, s represents the guard's internal state. With these definitions established, we are able to describe very different types of guards, utilizing different filters and interfaces. In the next section, we will first discuss different types of output filters applicable to operating system guards.

IV. Output Filter Types used in Guards

A. Filter Types identified

In the following we identify different output filter types used in operating system guards. The presented output filters can be applied to any guard $g \in G$. It should be noted that this presentation of filters is by no means meant to be exhaustive, as arbitrary operations could be carried out by ϕ. Furthermore, input filters are not presented here, but, for most output filters, there is also a corresponding input filter (e.g. a decryption filter for the encryption filter and an integrity evaluation filter for the integrity output filter). The filter types presented here were deducted from different problem statements we gathered in the process of designing operating system guards for both pure isolation and sharing hypervisors. As part of an operating system guard, some of these filters have been implemented in C++ code. The implementation results will be described in a future article.

B. Signal Distortion Output Filter

The signal distortion output filter distorts signals in a way that filters out certain signal characteristics, which should be semantically irrelevant for the actual transmitted content. An example for a signal distortion output filter would be a bandpass filter in a sound processing system that lets only audible sound frequencies pass to counter attacks based on modulated audio signals. Modulated audio signals could be used for establishing illegitimate information flows between operating partitions, i.e. establishing a *covert physical channel* (as opposed to covert storage channels and covert timing channels). We use the terminology of a covert physical channel here because covert channels are defined as channels that have not been designed for communication at all [26]. The covert physical channel could also be used as a network covert channel and for connections over multiple hops, extending the communication range of the channel and maybe maliciously connecting the computing system with further networks (e.g. the internet). In the operating system covert channel scenario,

if partition p_i was in control of the computer's speaker and partition p_j was in control of a microphone, data could be maliciously exchanged between p_i and p_j ($i \neq j$). In order to hide the covert physical channel from the computing system's user, the attacker could switch to inaudible frequencies (e.g. ultrasound). This specific covert channel type can effectively be prevented from utilizing inaudible signals by means of a signal distortion output filter applied to an audio filtering guard. An ultrasonic-based covert channel has been tested by Hanspach and will be described in a future article. Besides audio modulation/demodulation, other types of channels utilizing physical emanations are also conceivable, as Frankland [30] already presented an optical channel between keyboard LEDs and a camera, while Hasan et al. [31] presented physical approaches for near-field-communication command-and-control in botnets, utilizing different types of emanations.

C. Content Output Filter

The content output filter (Fig. 2) is a filter that separates the originally transmitted data into legitimate information and illegitimate information, whereby only legitimate information is allowed to pass through. Content filters could be used for

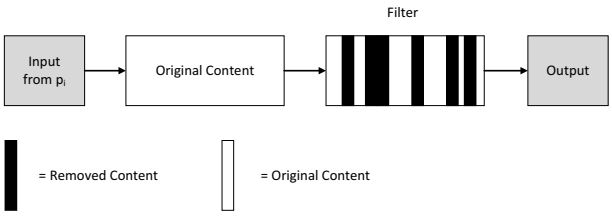

Fig. 2. Content Output Filter

deep packet inspection in application level gateways, filtering out any illegitimate content. Looking at a privacy context, content filters could be used to prevent accidental or malicious leak of personal data in personal computing devices (e.g. in smartphones). For operating systems in multilevel security mode, the content filter could be used to filter out classified data in document parts (often manually) marked as classified, even if the classified data was actually created in a domain not suitable for the targeted classification level. Moreover, the content filter could be used to prevent accidental information leaking. For instance, in documents created by common office software, text fragments are often blackened to prevent disclosure of critical information when handing over these documents. In a very common case, these modified documents are handed over electronically and, when the content of these text fragments is not actually deleted but only blackened, the information could be restored by the receiver of the document. For this purpose, the content filter could recognize such patterns of human mistakes by deep inspection of the transferred data and correct the presentation of the document. In a different filter implementation, the user could be informed of his accident, to grant him the chance to manually decide over the possible procedures applied.

D. Decision Output Filter

The decision output filter (Fig. 3) is applied to reach a decision whether an entire PDU (protocol data unit) should be discarded or forwarded. An example for a decision output

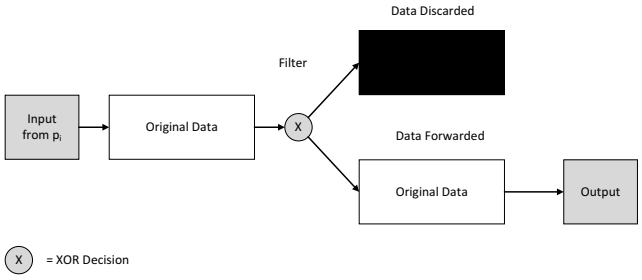

Fig. 3. Decision Output Filter

filter would be an operating system's packet filter, dropping illegitimate network packets or accepting them for processing or forwarding. The depicted basic form of the decision output filter might be extended by including an update procedure to the guard's internal state in order to support concepts like stateful packet inspection. The decision output filter can also be used to create a policy for component interconnection and isolation, connecting different partitions with a guard, but also isolating the partitions from communicating with each other on the middleware level.

E. Encryption Output Filter

The encryption output filter only forwards the ciphertext data $E(d)$ for a given plaintext data d, using a symmetric encryption algorithm for ϕ. As we want to separate our application partitions, different symmetric keys would have to be used for encryption/decryption request from different partitions. The encryption output filter can be used to implement a mandatory encryption policy, encrypting any processed data and preventing the forwarding of any unencrypted data. The encryption output filter might not only be used for security and privacy purposes, such as hard disk and network encryption but also for isolation purposes between distinct operating system partitions. Consider an example where two isolated partitions are connected to the same guard, accessing the same storage device. Without encrypting the stored data, the storage device (or storage device driver) could maliciously deliver critical data to the wrong partition, breaking the system's security and privacy policy. An encryption guard component has been suggested by Alves-Foss et al. [18], but we are utilizing encryption not solely for the purpose of data confidentiality against attackers on the wire or with physical access to H, but also for preventing storage channels inside the operating system by implementing a storage guard (as described in the next section).

F. Integrity Output Filter

The integrity output filter (Fig. 4) is designed to guard the integrity of output data. For every PDU processed, the integrity output filter forwards the original data, and stores a unique reference and an integrity anchor (i.e. a hash sum

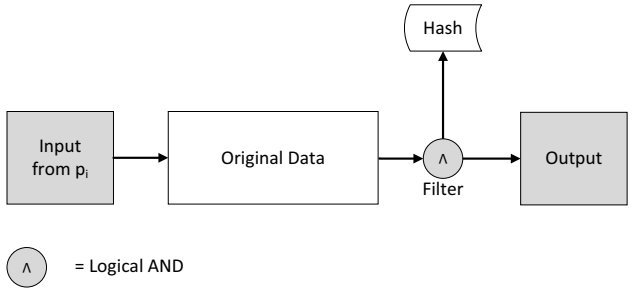

Fig. 4. Integrity Output Filter

or a message authentication code) inside the guard's internal state, which can be evaluated by a corresponding integrity input filter. The integrity filter is primarily designed for use in (local or distributed) storage subsystems, where data is first written out and needs to be reread at a later time in an integrity-preserving way.

G. Discard Output Filter

The discard output filter simply discards any output. The discard output filter can be used at any guard interface to build a data diode, only allowing the establishment of information flows in a specific direction. Data diodes may be used in *sharing guards*, which are utilized by a sharing hypervisor, ensuring that information flows are only established from higher to lower classification levels or from lower to higher trust levels (i.e. with the Bell-LaPadula or Biba model applied [25], [32]).

In the following section, we will take a look at the design of operating system guards and how they utilize the described filters to assure security and privacy for other components and application partitions.

V. A Typology of Guards applicable in component-based Operating Systems

A. Types of Guards

We will first describe the different types of guards we consider in this article. This collection of guards is, just as the collection of output filters, not meant to be exhaustive, as guards are defined by their application of input and output filters and arbitrary filters could be defined.

The *storage guard* fulfills the purpose of preventing storage channels between any pair p_i/p_j of different application partitions in a scenario with a shared storage device. The storage guard has to partition the shared storage device into different isolated sections, enforcing the policy that each section shall be isolated from access of a non-associated p_i. Furthermore, the storage guard has to ensure that only encrypted data is written to the shared storage device, so that no p_i could read out data from p_j, even in the case of hardware manipulations or failures. Finally, the storage guard has to verify the integrity output data, applying integrity input and output filters.

The *audio filtering guard* fulfills the purpose of preventing storage channels between different operating system partitions. The audio filtering guard implements signal distortion output or input filters to prevent any audio output from a p_i that utilizes inaudible audio frequencies. Generalizing the audio filtering guard approach, we could construct hardware I/O filtering guards for any type of I/O devices.

The *downgrading guard* is used to prepare documents for downgrading from a higher to a lower classification in order to enable information exchange from higher to lower classification levels in a sharing hypervisor. A downgrader for multilevel security systems is also suggested by Alves-Foss et al. [18].

The *content filtering guard* serves the purpose to filter out illegitimate content in PDUs, for instance to filter out blackened data in documents, and applies the content output and input filters, preventing p_i and c from accidental or malicious establishment of illegitimate information flows.

The *sharing guard* allows for data sharing in a sharing hypervisor (as opposed to a pure isolation hypervisor [24]), implementing multilevel security policies like the Bell-LaPadula and the Biba model [25], [32].

The *policy guard* can be used to implement access control policies for interconnection between components where the access control policy needs to be altered at runtime and a mandatory access control policy would be too inflexible. The general concept can also be used to delegate IPC tasks from the micro kernel to policy guards in order to reduce the complexity of the kernel policies and delegate different policy decisions to different system roles. Note that the micro kernel's reference monitor (governing all information flows in the operating system as explained by Anderson [10]) would still be always-invoked by the trusted policy guard, but the actual decision would be delegated to the guard to the extent of all possible information flows that are granted to the guard. As a prior example of a policy guard, Alves-Foss et al. [18] introduced the MILS Message Router that "will function as a data switch by taking data from multiple partitions at various classification levels and routing the messages to the correct destination, which may include additional trusted devices that determine if the message satisfies the application-level security policy". Another example of a delegating policy guard has been presented by Payne et al. [33].

The *sequencing guard* is a specific variant of the policy guard and can be used to interconnect different applications with operating system components performing the processing in a fixed order, analogous to the fixed processing order of an assembly line. The sequencing guard can, for instance, be used for protocol data assembling, adding different security protocol headers in a predetermined sequence in line with the security protocol definition.

By the definition of an operating system guard (see Section III), a guard can utilize one or more filters in order to protect other operating system components and application partitions. In Tab. I, a mapping between the above described operating system guards and their utilized output filters is established. It can be seen in this mapping that we can create new types of operating system guards by application of different filter types. It is also made visible that decision output filters should normally be applied to guards in order to prevent illegitimate communication between partitions over

TABLE I. MAPPING BETWEEN GUARDS AND FILTERS

Guards / Filters	Signal Distortion Output Filter	Content Output Filter	Decision Output Filter	Encryption Output Filter	Integrity Output Filter	Discard Output Filter
Storage Guard			X	X	X	
Audio Filtering Guard	X		X			
Downgrading Guard		X	X			
Content Filtering Guard		X	X			
Sharing Guard			X			X
Policy Guard			X			X
Sequencing Guard			X			X

the operating system guard interfaces. To get a more precise impression of the construction of operating system guards, we will now have closer look at three distinct types of guards.

B. A closer look at the storage guard

The purpose of the storage guard (Fig. 5) is to provide storage access to isolated different application partitions, but using a shared storage device (e.g. an HDD or SSD). The storage guard would not only be utilizable for security but also for privacy applications, completely isolating partitions from different users on a shared storage device. Because we are using a shared storage device, we have to ensure that no unencrypted data is ever stored on the storage device, to prevent the storage device from maliciously returning data belonging to p_j to p_i. Therefore, encryption and decryption filters have to be applied at the interfaces between the guard and the storage device (or perhaps the storage device driver as an intermediate), implementing a red/black (unencrypted/encrypted) separation inside the guard. Different encryption/decryption keys have to be used for requests from different partitions, to prevent any partition from accessing data that was encrypted for a different partition.

A malicious storage device or device driver could further try to attack the integrity of the stored data by returning manipulated data, making integrity-preserving precautions necessary. For this purpose, the storage guard would implement integrity checks at the interfaces to the storage device or the device driver. For preventing any communication between p_i and p_j over the guard, decision output filters would have to be applied. In this example, IF_{in} and IF_{out} would contain the interfaces to p_i, p_j and H. $F_{in} \cup F_{out}$ would contain encryption/decryption filters and integrity input/output filters at the interface to H. At the interfaces to p_i and p_j, a decision output filter would be implemented as described above. By application of the decomposition refinement pattern [27] to the storage guard, we can further advance the storage guard by outplacing the hardware access functionalities (i.e. the storage driver) into an untrusted component (e.g. a legacy driver for maximized compatibility), only concentrating on security measures inside the storage guard and maximizing the compatibility with existing source code. Although we used an example of two connected application partitions, any number of application partitions could be connected with the storage guard to access the shared storage device. The storage guard has been implemented by us as a prototype of a C++ application and will be described in a future article.

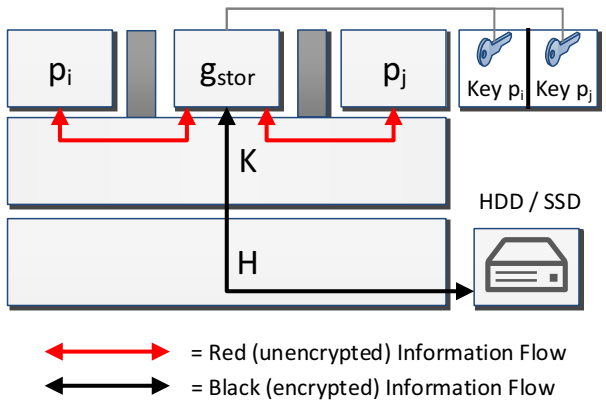

Fig. 5. A storage guard used by two different application partitions

C. A closer look at the audio filtering guard

The audio filtering guard (Fig. 6) connects any number (two in this example) of p_i partitions with a shared sound processing hardware.

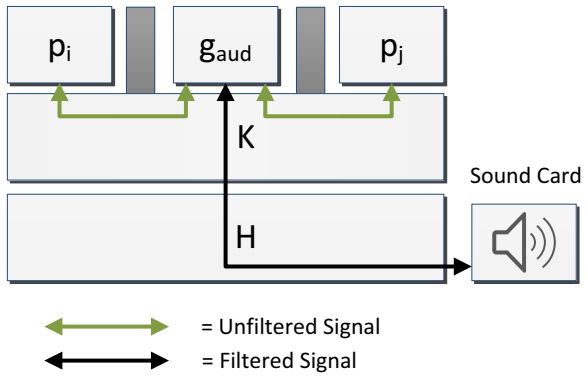

Fig. 6. An audio filtering guard used by two different application partitions

IF_{in} and IF_{out} would contain the interfaces to p_i and p_j, providing audio services and audio filtering to both of partitions. F_{out} (and optionally F_{in}) would contain signal distortion filters, utilizing the input or output interfaces between p_i and g_{aud} to prevent any output in the ultrasound frequency spectrum. This way, ultrasonic covert channels between operating system partitions (and between different computers) could effectively be prevented. Ultrasonic covert channels are actually a considerable problem in high-assurance computing systems as we found out in multiple experiments. We will describe our experiments on ultrasonic covert channels in detail in a future article. F_{out} would also (just as in the storage guard) contain decision output filters at the interfaces to p_i and p_j to prevent communications between p_i and p_j over the audio filtering guard. Just as with the storage guard, refinement patterns like decomposition could be applied in order to reutilize legacy audio drivers.

D. A closer look at the sequencing guard

The sequencing guard (Fig. 7) is used to perform different operations, implemented in different components in a specific order. The numbers depicted show the logical processing order position in the sequence $p_i \rightarrow g_1 \rightarrow g_2 \rightarrow g_3$ where any transition is enabled by the sequencing guard g_0.

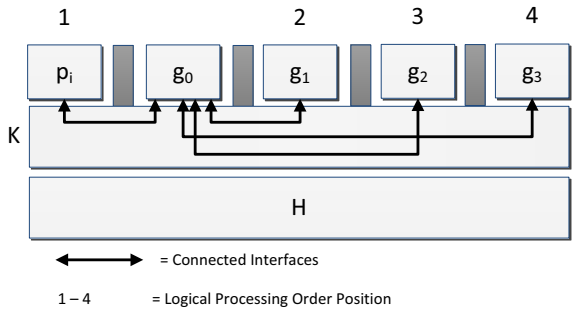

Fig. 7. A sequencing guard, utilizing three processing components

IF_{in} and IF_{out} of the sequence guard would contain the input and output interfaces to the processing components g_1 and g_2, the original input partition p_i and the final output component g_3. F_{in} and F_{out} would contain decision input/output filters and discard output filters to manage the information flow and enforce the defined processing order. The sequencing guard also needs to keep track of a PDU's processing history in its internal state in order to determine the next processing step, which is enforced by the decision-based filters. The sequencing guard could be applied where different filtering steps, implemented in different operating system guards, have to be processed in a specific order in order to pass the security check. As a possible application, the sequencing guard could be implemented for performing separated processing steps in a network packet filter.

E. The guard reference architecture

From the architecture of the presented guards, we can deduct a general guard reference architecture (Fig. 8). The guard reference architecture can be conceived as a general design pattern for the development of operating system guards, which did not exist before to the best of our knowledge. The guard reference architecture involves potential interfaces between the guard g and any partition p, c, g and to K and H. All of these information flows are governed by K, implementing the reference monitor paradigm [10], implying that the kernel has to be always-invoked in IPC. The guard reference architecture can be conceived as the prototype of an *n-way-guard* [33], connecting n different elements of P, G and C, and K and H and applying specific input and output filters for a specific purpose. With the guard reference architecture and the mathematical guard notation defined, we should be able to develop a general *guard construction kit* that could be implemented as a GUI-based software, only choosing the associated input and output interfaces, and corresponding input and output filters and the guard's initial state for the creation of a new guard.

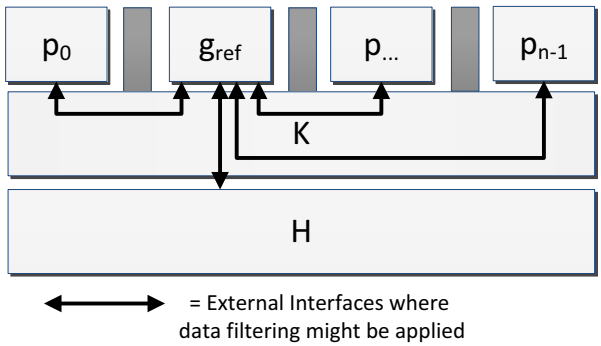

Fig. 8. The suggested guard reference architecture

VI. Conclusion

Guards protect critical operations in trusted parts of the operating system from attacks of possibly infected, untrusted parts. Guards are trusted components that serve multiple protection purposes in component-based operating systems, in multilevel security systems and, especially with the introduction of separation VMMs, even in commodity operating systems. Guards can, for instance, serve as a multiplexer component, interconnecting isolated operating system partitions with distinct parts of a shared resource and, thereby, also partitioning the shared resource. Guards can also serve as a policy service, enforcing system policies on behalf of the kernel. And, what's more, guards can prevent application partitions from illegitimate output and covert communications by filtering out unexpected contents or signal characteristics. All the operating system's guards taken together form a trusted middleware that provides the application partitions with trusted services like encryption, which are not provided by the kernel as an evaluatable and verifiable component (and explicitly shall not be provided in a stripped-down micro kernel).

We discussed a typology of different guards representing different compilations of input and output interfaces, and input and output filters. The storage guard, the audio filtering guard and the sequencing guard have been described in detail. While the storage guard is designed to counter Lampson's storage channels and is designed to counter both security and privacy escalations, the audio filtering guard allows for the prevention of physical covert channels, utilizing audio communications in the ultrasound frequency range. From our definitions and the examples provided, we deduct the reference architecture of an operating system guard, allowing us to simply create new guards just by definition of a small number of variables. And, because of these formal definitions, we are able to harmonize the construction of new operating system guards and develop the concept of a *guard construction kit*, which offers new opportunities for research on operating system guards. As the filtering functions and the state modification procedures in an operating system guard could be chosen arbitrarily, any number of different guards could be defined and no compilation (including this one) could ever be exhaustive. Still, with our compilation of operating system guards, we have been targeting some of the common security problems that

are found in the literature. Beside implementation results in regard to operating system guards, future work might include approaches to formal verification of both implementation and formal specification of an operating system guard.

ACKNOWLEDGMENT

We would like to thank Alexander Senier for helpful discussions on the storage guard.

REFERENCES

[1] T. Jaeger, J. Liedtke, V. Panteleenko, Y. Park, and N. Islam, "Security architecture for component-based operating systems," in *Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, ser. EW 8. New York, NY, USA: ACM, 1998, pp. 222–228.

[2] G. Heiser, L. Ryzhyk, M. Von Tessin, and A. Budzynowski, "What if you could actually trust your kernel?" in *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, ser. HotOS'13. Berkeley, CA, USA: USENIX Association, 2011, pp. 27–27.

[3] G. Klein, "The L4.verified project: next steps," in *Proceedings of the Third international conference on Verified software: theories, tools, experiments*, ser. VSTTE'10. Berlin, Heidelberg: Springer, 2010, pp. 86–96.

[4] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: formal verification of an OS kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 207–220.

[5] G. Klein, T. Murray, P. Gammie, T. Sewell, and S. Winwood, "Provable Security: how feasible is it?" in *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, ser. HotOS'13. Berkeley, CA, USA: USENIX Association, 2011, pp. 28–28.

[6] J. Andronick, D. Greenaway, and K. Elphinstone, "Towards proving security in the presence of large untrusted components," in *Proceedings of the 5th international conference on Systems software verification*, ser. SSV'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 9–9.

[7] J. Andronick, "From a proven correct microkernel to trustworthy large systems," in *Proceedings of the 2010 international conference on Formal verification of object-oriented software*, ser. FoVeOOS'10. Berlin, Heidelberg: Springer, 2011, pp. 1–9.

[8] C. L. Heitmeyer, M. Archer, E. I. Leonard, and J. McLean, "Formal specification and verification of data separation in a separation kernel for an embedded system," in *Proceedings of the 13th ACM conference on Computer and communications security*, ser. CCS '06. New York, NY, USA: ACM, 2006, pp. 346–355.

[9] S. Chong and R. van der Meyden, "Deriving epistemic conclusions from agent architecture," in *Proceedings of the 12th Conference on Theoretical Aspects of Rationality and Knowledge*, ser. TARK '09. New York, NY, USA: ACM, 2009, pp. 61–70.

[10] J. P. Anderson, "Computer Security Technology Planning Study," *Tech. Rep. ESD-TR-73-51, Volume II. Electronic Systems Divison, AFSC*, Oct. 1972.

[11] J. M. Rushby, "Design and verification of secure systems," *SIGOPS Oper. Syst. Rev.*, vol. 15, no. 5, pp. 12–21, Dec. 1981.

[12] J. McDermott, B. Montrose, M. Li, J. Kirby, and M. Kang, "Separation virtual machine monitors," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC '12. New York, NY, USA: ACM, 2012, pp. 419–428.

[13] P. G. Neumann, "System and network trustworthiness in perspective," in *Proceedings of the 13th ACM conference on Computer and communications security*, ser. CCS '06. New York, NY, USA: ACM, 2006, pp. 1–5.

[14] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "InkTag: secure applications on an untrusted operating system," in *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 265–278.

[15] J. A. Solworth, "Robustly secure computer systems: a new security paradigm of system discontinuity," in *Proceedings of the 2007 Workshop on New Security Paradigms*, ser. NSPW '07. New York, NY, USA: ACM, 2008, pp. 55–65.

[16] S. M. Bellovin, "Virtual machines, virtual security?" *Commun. ACM*, vol. 49, no. 10, pp. 104–, Oct. 2006.

[17] S. Bratus, M. E. Locasto, A. Ramaswamy, and S. W. Smith, "VM-based security overkill: a lament for applied systems security research," in *Proceedings of the 2010 workshop on New security paradigms*, ser. NSPW '10. New York, NY, USA: ACM, 2010, pp. 51–60.

[18] J. Alves-Foss, W. S. Harrison, P. Oman, and C. Taylor, "The MILS architecture for high-assurance embedded systems," *International Journal of Embedded Systems*, vol. 2, pp. 239–247, 2006.

[19] J. Jacob, "MILS: High-Assurance Security at Affordable Costs," *The Journal of Military Electronics & Computing*, Nov. 2005.

[20] A. G. Camek, C. Buckl, and A. Knoll, "Future cars: necessity for an adaptive and distributed multiple independent levels of security architecture," in *Proceedings of the 2nd ACM international conference on High confidence networked systems*. New York, NY, USA: ACM, 2013, pp. 17–24.

[21] C. Partridge, R. Walsh, M. Gillen, G. Lauer, J. Lowry, W. T. Strayer, D. Kong, D. Levin, J. Loyall, and M. Paulitsch, "A secure content network in space," in *Proceedings of the seventh ACM international workshop on Challenged networks*, ser. CHANTS '12. New York, NY, USA: ACM, 2012, pp. 43–50.

[22] C. Li, A. Raghunathan, and N. K. Jha, "An architecture for secure software defined radio," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '09. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2009, pp. 448–453.

[23] T. E. Levin, C. E. Irvine, C. Weissman, and T. D. Nguyen, "Analysis of three multilevel security architectures," in *Proceedings of the 2007 ACM workshop on Computer security architecture*, ser. CSAW '07. New York, NY, USA: ACM, 2007, pp. 37–46.

[24] P. A. Karger, "Multi-Level Security Requirements for Hypervisors," in *In Proceedings of the 21st Annual Computer Security Applications Conference (December 05 - 09, 2005). ACSAC. IEEE Computer Society*, 2005, pp. 5–9.

[25] D. E. Bell and L. J. LaPadula, "Secure Computer Systems: Mathematical Foundations," *The MITRE Corporation, Tech. Rep. 2547*, vol. I, Mar. 1973.

[26] B. W. Lampson, "A note on the confinement problem," *Commun. ACM*, vol. 16, no. 10, pp. 613–615, Oct. 1973.

[27] J. Zhou and J. Alves-Foss, "Architecture-based refinements for secure computer systems design," in *Proceedings of the 2006 International Conference on Privacy, Security and Trust: Bridge the Gap Between PST Technologies and Business Services*, ser. PST '06. New York, NY, USA: ACM, 2006, pp. 15:1–15:11.

[28] J. C. Robinson and J. Alves-Foss, "A high assurance MLS file server," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 1, pp. 45–53, Jan. 2007.

[29] J. C. Robinson, W. S. Harrison, N. Hanebutte, P. Oman, and J. Alves-Foss, "Implementing middleware for content filtering and information flow control," in *Proceedings of the 2007 ACM workshop on Computer security architecture*, ser. CSAW '07. New York, NY, USA: ACM, 2007, pp. 47–53.

[30] R. Frankland, "Side Channels, Compromising Emanations and Surveillance: Current and future technologies," Department of Mathematics, Royal Holloway, University of London, Egham, Surrey TW20 0EX, England, Tech. Rep. RHUL-MA-2011-07, Mar. 2011.

[31] R. Hasan, N. Saxena, T. Haleviz, S. Zawoad, and D. Rinehart, "Sensing-enabled channels for hard-to-detect command and control of mobile devices," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, ser. ASIA CCS '13. New York, NY, USA: ACM, 2013, pp. 469–480.

[32] K. J. Biba, "Integrity Considerations for Secure Computer Systems," *The MITRE Corporation, Tech. Rep. ESD-TR 76-372*, May 1977.

[33] B. D. Payne, R. Sailer, R. Cáceres, R. Perez, and W. Lee, "A layered approach to simplified access control in virtualized systems," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 4, pp. 12–19, Jul. 2007.