A Formal Model for Virtual Machine Introspection

Jonas Pfoh, Christian Schneider, and Claudia Eckert
Department of Computer Science
Technische Universität München
Munich, Germany
{pfoh,schneidc,eckertc}@in.tum.de

ABSTRACT
Virtual machine introspection (VMI) describes the method of monitoring and analyzing the state of a virtual machine from the hypervisor level. In this paper, we present a formal discussion of the development of VMI-based security applications. We begin by identifying three major challenges that all VMI-based security applications must overcome. The main contribution of our work is the definition of a formal model for describing VMI techniques. This model is broken down in such a way that allows for thorough discussion of any VMI approach with regard to each of the three challenges. Then, we specify three design patterns for interpreting state information using our model. We argue that these patterns are complete, that is, they cover all possible methods for state interpretation. The properties of all patterns are thoroughly discussed so that the pros and cons of their application may be fully understood. Finally, we describe and discuss an ideal VMI-based intrusion detection system using our model and begin to detail the practical implications in building such a system.

Categories and Subject Descriptors
D.4.6 [Operating Systems]: Security and Protection—Invasive Software; H.1.0 [Models and Principles]: General

General Terms
Security

Keywords
Virtualization, Security, Intrusion Detection, Formalization, Introspection

1. INTRODUCTION
System virtualization provides an interface to a virtual hardware platform implemented in software. This software layer is called a hypervisor or a virtual machine monitor (VMM). The hypervisor allows an operating system (OS) to run as a guest in a virtual machine (VM) while maintaining control of the physical resources (e.g., CPU, RAM, and devices). Such an architecture inherently gives the hypervisor a complete and untainted view of all system state information. This system state is comprised of CPU and I/O register values as well as volatile and stable system storage contents. Such a view provides the ability to thoroughly observe and analyze the guest operating system from outside. Garfinkel and Rosenblum [4] have coined the term virtual machine introspection (VMI) to describe such virtualization assisted monitoring and analysis.

Security mechanisms such as digital forensic tools and host-based intrusion detection systems (IDSs) rely on the ability to observe the system state. As such, these mechanisms clearly benefit from the properties of VMI. For example, performing a forensic analysis of the guest’s volatile memory will not affect the state of the guest. If this type of analysis were to be done without VMI, the analyst would need to consider the effects of his analysis on the state of the machine. Similarly, a host-based IDS no longer needs to rely on information provided by an operating system which cannot be trusted if the host is compromised.

In this paper, we explore the challenges that must be overcome when leveraging VMI to its full potential, and we introduce a formal model which helps in examining and reasoning about possible VMI approaches. We focus on leveraging VMI to perform intrusion detection on the OS level, i.e., detection of system compromises with administrative privileges which alter the OS state in such a way that the intruder may remotely access the system, abuse system resources, or collect information from the system or its users. These types of attacks include but are not restricted to rootkits. User-level compromises that do not affect the operating system state are out of the scope of our approach.

While we focus on intrusion detection, the arguments made in this paper are applicable to further security applications of VMI (e.g., forensics, secure logging) as well.

The remainder of this paper is organized as follows. In the next chapter we give a brief introduction of the features of VMI and identify the three main challenges for VMI-based security applications. We then present our formal model in Section 3. With this foundation, we derive the three view-generation patterns in Section 4. A way of approaching a VMI-based intrusion detection system follows in Section 5. In Section 6, we explore some related work and finally conclude the paper in Section 7.
2. VIRTUAL MACHINE INTROSPECTION

Introspection is a synonym for self-examination. Virtual machine introspection describes the act of examining and monitoring the virtual machine (i.e., the state of the guest OS) from the vantage point of a VMM or a second privileged guest OS. As discussed in the introduction, the VMM naturally allows for this as all system state is inherently visible.

In [4], Garfinkel and Rosenblum lay the foundation for this type of work and present the huge potential that VMI has, not only for intrusion detection, but also for other fields within IT security. They discuss those properties of virtualization that support intrusion detection from the hypervisor level, namely isolation, inspection, and interposition. Isolation refers to the property that (ideally) the VMM cannot be tampered with. Inspection refers to the property that allows the VMM to examine the entire state of the guest OS without having to rely on a possibly compromised system for this information. Finally, Interposition refers to the ability to inject operations into the normal running of the system based on certain conditions. These properties lead to the following advantages of VMI which are beneficial for many security applications:

- A complete and untainted view of the system state.
- A consistent view of the system state, since the VM may be paused.
- The ability to manipulate the VM state.
- Complete isolation of the VMI mechanism from the guest OS.

In order to leverage the full potential provided by these advantages, the following challenges must be overcome:

1. Interpreting the immense amount of binary low-level data that comprise the system state to obtain abstract high-level state information.
2. Identifying relevant parts of that state information for a given application.
3. An application-appropriate classification of both unknown and known system states.

To be able to rely on the advantages that VMI brings, the VMM as well as any privileged VMs involved in the VMI must not be vulnerable to compromise. Due to the strong isolation properties provided by the VMM, the number of attack vectors on the external monitoring components is significantly reduced compared to software running inside the monitored VM.

3. VMI MODEL

In order to overcome the previously mentioned challenges, it becomes beneficial to use a model for describing and comparing VMI approaches.

3.1 States, Views, and Classes

All VMI approaches inspect guest system states. A guest system state is comprised either of all the guest system information that is visible to the hypervisor or the guest OS, i.e., CPU registers, RAM, stable storage, and connected devices. For our model, we begin by defining two sets of states:

Definition 1 We define $S_{int} \subseteq \{0,1\}^n$ as the set of all possible VM states visible to the guest OS being monitored, and we define $S_{ext} \subseteq \{0,1\}^m$ as the set of all possible VM states visible to the VMM. A single state is represented as a finite length string of bits, where $n$ and $m$ are the number of bits required to represent an entire state from a respective vantage point.

A state is simply the concatenation of all visible values within the VM. Concretely, a state is the concatenation of all CPU register values, with the entire contents of memory, with the entire contents of stable storage, and so on.

One challenge of creating a VMI-based security mechanism is classifying any given machine state based on the specific application. For example, an intrusion detection system might classify a given state as “compromised” or “uncompromised”. This is formalized in the following definition:

Definition 2 We define $C$ as the set of all possible classifications for machine states for a specific scenario.

Given a machine state $s \in S$, ideally one would like to have a function $f : S \rightarrow C$ that directly outputs the correct classification for $s$. However, this may not be possible in practice. Since a system state contains lots of information that is irrelevant to the classification, this irrelevant information must be filtered out. For example, modern OSs tend to reserve a large amount of memory to cache data from stable storage. This cache represents a portion of memory which has no valuable contribution for many security mechanisms, such as intrusion detection.\(^1\)

Additionally, some $s, s' \in S$ with $s \neq s'$ may represent an equivalent abstract system state, but differ in their binary representation due to the randomness in a running kernel, such as memory locations of dynamic data structures, i-nodes, process IDs, etc. For example, the order in which processes appear in the process list may not be important from a system security perspective. Consequently, all permutations of a process list would be considered equivalent.

In order to represent these factors in our model, we introduce the notion of a view:

Definition 3 We define $V_{int}$ and $V_{ext}$ as the set of all possible views constructible from $S_{int}$ and $S_{ext}$, respectively. A view is a concise representation of the state that combines the result of zero or more of the following operations on the input string (i.e., an element of $S_{int}$ or $S_{ext}$):

1. The view may be a projection of the input string, that is, a subsequence of the input.
2. The view may be a permutation of the input string. Generally, fixed length substrings representing internal data structures are permuted.
3. The view may be a compression of the input string, that is, an interpretation of the input string based on knowledge of the system architecture.

\(^1\)In fact, there do exist viruses that infect disk caches, e.g., Darth\_Vader or W95\_Repus, but due to the inefficiency of their replication method, such viruses are not prevalent in the wild.
We define a virtual hardware architecture description $\lambda$ and a guest system software architecture description $\mu$, we define the following functions:

$$
\begin{align*}
\lambda: S_{\text{int}} &\to V_{\text{int}} & \text{(Internal view-generation)} \\
\mu: S_{\text{ext}} &\to V_{\text{ext}} & \text{(External view-generation)} \\
a : V_{\text{int}} \times V_{\text{ext}} \times P &\to P & \text{(Aggregation)} \\
d : P &\to C & \text{(Classification)}
\end{align*}
$$

We refer to the view-generating functions as internal and external as well as use the subscripts $\text{int}$ and $\text{ext}$ for the sets of states and views. We may also refer to a process that is happening internally or externally. These terms reflect the view point of the monitored VM. That is, “internal” refers to a function or process that occurs within the monitored VM, while “external” refers to a function or process that occurs within the VMM or a privileged VM.

The flow of information in the model is depicted in Figure 1. The view-generating function $f_{\lambda,\mu}$ constructs a view internally, while $g_{\lambda,\mu}$ constructs a view externally. The aggregation function $a$ takes these views along with the current profile to create a new profile of the system state. This profile is, in turn, processed by the classification function $d$ to classify the current system state. Both the aggregation function and the classification function are executed externally.

The subscripts that the view-generating function labels carry is a pair $\lambda, \mu$, which denotes that each function constructs its output based on $\mu$ and $\lambda$. For a particular VMI approach, it is possible that either $\mu = \emptyset$ or $\lambda = \emptyset$ (i.e., an approach which makes use of a view-generating function that constructs its output based solely on the knowledge of either $\mu$ or $\lambda$). Going forth, rather than explicitly stating that either $\mu = \emptyset$ or $\lambda = \emptyset$, we simply leave the appropriate subscript out of the notation. For example, we denote the function $f_{\lambda,\mu}$, where $\lambda = \emptyset$ as $f_{\mu}$.

Additionally, it is possible that for a particular VMI approach either $f_{\lambda,\mu}$ or $g_{\lambda,\mu}$ always outputs $\varepsilon$, where $\varepsilon$ is the empty string. In a practical sense, this represents the lack of a corresponding component in the implementation. For example, a VMI approach may only contain a view-generating component within the hypervisor. In order to denote this, we simply omit the appropriate view-generating function as...
well as the appropriate input into the aggregation function when describing such an approach.

3.3 Purpose

This model breaks a VMI system into its basic components and their interactions, thus reflecting important design decisions. This is very helpful for understanding and reasoning about a VMI system design. As we will show in Section 4, there are different ways to reach the same goal in a VMI system which result in different security, safety, and portability properties. Taking the time to describe a system with our model forces one to look at all aspects of the system design. In doing so one may find a flaw in parts of the system that were not the focus of the initial design. However, it is important to note that this model is *not* meant as a means for formal verification.

Our model reveals properties for individual view-generating functions in a fine-grained manner. These properties are explained in Section 4. The utility of our model becomes especially evident in cases where two functions result in contradicting properties, e.g., a hardware portable function in combination with a function which is not hardware portable. Considering how these contradicting properties interact and coming up with the ensuing property of the system as a whole takes some thought. This thought process is easily overlooked or taken for granted in some circumstances. A further discussion on this issue can be found in Section 4.4. If a design does not conform to the properties it was set out to achieve, it may be tweaked and re-checked.

We have found this model to be a very helpful tool in our work and use it throughout the remainder of this paper.

4. BRIDGING THE SEMANTIC GAP

From the vantage point of the hypervisor, the guest system state is seen in a strictly binary form. Creating a view in this situation becomes quite challenging due to the fact that the external view-generating function $g_{\lambda,\mu}$ has no native knowledge about what this binary data represents, for example, which data structures lie where in memory. Chen and Noble [3] refer to this challenge as the “semantic gap”.

Bridging this semantic gap is a matter of view generation. That is, discussing the different methods of generating views is, in fact, a discussion of ways to bridge the semantic gap. This is an excellent example of how our model aids in the discussion of a particular aspect of VMI.

In order for the hypervisor-based view-generating function $g_{\lambda,\mu}$ to create an appropriate view where the semantics of every bit is known, the function needs to make use of knowledge about either the guest system software architecture $\mu$, the virtual hardware architecture $\lambda$, or a combination of the two. We refer to this knowledge as *semantic knowledge* because it gives the function knowledge about the meaning of the observed system state.

Using our model, we present three view-generation patterns that may be used alone or in any combination as a means for bridging the semantic gap, namely: the *out-of-band delivery* pattern, the *in-band delivery* pattern, and the *derivative pattern*. They differ in two ways, first, where within the architecture the view-generation takes place (i.e., internally or externally) and, second, how semantic knowledge is incorporated. They can then be used alone or in combination to describe all possible methods for bridging the semantic gap. What follows is an in depth discussion of each pattern. This discussion will often refer to certain functions within the VMM. By this we only mean that they are executed externally. Such functions can just as well be carried out within a separate, privileged VM without affecting our discussion.

4.1 Out-of-Band Delivery

This pattern is the most commonly used method for bridging the semantic gap. Here semantic knowledge is *delivered* to the external view-generating function in an *out-of-band* manner. That is, the external view-generating function receives the semantic knowledge in advance, before VMI begins. For example, the VMM may make use of a previously delivered symbol table based on the guest OS kernel to determine the position of key data structures. Formally, this pattern is described as follows:

$$
\begin{align*}
g_{\mu} & : S_{ext} \rightarrow V_{ext} \\
\lambda & : V_{ext} \times P \rightarrow C \\
d & : P \rightarrow C
\end{align*}
$$

From this description, it can be seen that the view-generating function makes sole use of knowledge of the software architecture. Since this knowledge is built into the function, this function is only fit to create a view based on a software architecture for which $\mu$ is the description. If a malicious entity changes the software architecture (e.g., by inserting some hidden data structure), thus changing the appropriate description from $\mu$ to $\mu'$, clearly $g_{\mu}$ is no longer fit to provide an accurate view. The semantic knowledge (i.e., the software architecture description) that the view-generating function uses is in no way bound to the actual software architecture of the running guest. For this reason, such an approach is called “non-binding” [12].

Very closely related to the non-binding nature of this pattern is the lack of guest portability. That is, if the guest system software is changed or replaced completely, the software architecture description must also be changed to reflect this and the view-generating function must be constructed anew to accommodate the new system software.

As seen in the formal description, this pattern makes use of the external view-generating function, $g_{\mu}$. The fact that the view generation is performed externally results in three properties of this pattern. First, the function has sight of the complete system state. That is, every bit that makes up the system state is visible to an external function, even areas which are not accessible from within the VM. Second, the function is isolated from the guest. Therefore, any malicious entities which may have compromised the guest OS can not influence or corrupt view generation. Additionally, the component that implements this function will not interfere with the state of the guest, that is, there is no observer effect. Third, the VM does not need to be running while the view generation takes place. This results in a consistent view of the state. In addition, this allows one to mitigate external effects of keeping a compromised system running. For example, it is beneficial to suspend a system which acts hostile to neighbors on a production network while still being able to examine it.

This pattern is easily applied and implemented when compared to the derivation pattern, which is detailed below. In a practical example, monitoring any kernel-level data structure externally requires delivering the address and the layout of the data structure to the monitoring component. This in-
4.2 In-Band Delivery

The in-band delivery pattern describes an approach in which an internal component creates a view and delivers this view to the VMM. Since the view-generating function is internal, it may make use of the guest OS’s inherent knowledge of the software architecture. That is, semantic knowledge is being delivered in an in-band manner.

Upon closer inspection, this pattern does not so much bridge the semantic gap, but rather avoids it. In fact, one could rightfully argue that this is not a method for bridging the semantic gap in the purest sense since the view is generated from within the monitored guest OS. However, we include this pattern for the sake of completeness because it can be successfully combined with other patterns. This method is formally described as follows:

\[
\begin{align*}
    f_\lambda &: S_{int} \rightarrow V_{int} \\
    a &: V_{int} \times P \rightarrow P \\
    d &: P \rightarrow C
\end{align*}
\]

Having an internal view-generating function results in a few disadvantages. First, since view generation takes place within the monitored guest OS, the component that implements view generation is susceptible to compromise from any malicious entity that has compromised the monitored guest OS. In essence, it is necessary to trust a component which may be compromised. Second, an internal function does not support the ability to examine a suspended system, creating possible inconsistencies and resulting in the inability to mitigate external effects. Third, an internal function will always interfere with the system being monitored and finally, there may be parts of the state not visible to the view-generating function.

As with the out-of-band delivery pattern, this pattern is non-binding. This pattern also shares the lack of guest portability with the out-of-band delivery pattern. Both of these properties have been discussed in Section 4.1.

It seems appropriate to have a short discussion about why the in-band delivery pattern is not being considered with a view-generating function that makes use of hardware knowledge (i.e., \( f_{\lambda,\mu}, \lambda \neq \emptyset \)). While we admit it is practically possible to create such a view-generating function within the monitored guest OS, view generation using knowledge of the hardware architecture is always best done externally for one primary reason: Any view-generating function that makes use of hardware architectural knowledge can be implemented from within the VMM and gains all of the benefits associated with such an approach (i.e., isolation, consistency, etc.). On the other hand, implementing a view-generating function in the guest OS brings no advantages over an approach in which the function is implemented externally. For this reason, we argue that while implementing such a function internally is practically possible, there is absolutely no added value and is therefore not discussed further.

4.3 Derivation

The final VMI pattern moves away from the more common approaches and has the VMM derive information through semantic knowledge of the hardware architecture. For example, understanding a particular architecture and monitoring control registers within a CPU provides us with some semantic knowledge. Formally, this approach is described as follows:

\[
\begin{align*}
    g_\lambda &: S_{ext} \rightarrow V_{ext} \\
    a &: V_{ext} \times P \rightarrow P \\
    d &: P \rightarrow C
\end{align*}
\]

From our formal description, it is seen that this pattern makes use of the external view-generating function \( g_\lambda \), thus leading to the same advantages that are discussed in Section 4.1 with respect to the externality.

Further, it can be seen that \( g_\lambda \) constructs its output based on knowledge of the hardware architecture alone. This results in two additional advantages. First, this pattern is completely guest portable as the view-generating function makes no use of the guest software architecture. Second, this is a binding approach in contrast to the two delivery methods. That is, the hardware architecture description is bound to the hardware architecture of the virtual machine because there is no way for a malicious entity that compromised the guest OS to change the virtual hardware architecture. A malicious entity has no way of changing \( \lambda \), and whatever changes it may apply to \( \mu \) will not effect \( g_\lambda \)'s ability to create a view.

This seems like the ideal pattern, though in practice there is a large drawback to such an approach. The view that is generated is generally very constrained. A lot of information that is present in the state is lost through the view-generating function. This is due to the fact that there is information that cannot be extracted by only making use of hardware architectural knowledge and that information which can be extracted often needs much expertise and effort to be extracted.

Finally, this approach is obviously not hardware portable. If the view-generating function is built to interpret the state of a particular hardware architecture, then this function will clearly not be suited for use with another hardware architecture.

4.4 Combination of Patterns

The patterns that have been discussed up to this point are in their purest form. It is, of course, possible to combine such patterns. In fact, it is sometimes beneficial to use a drawback of one pattern in combination with another pattern to one’s advantage: A component of rootkit detection often involves trying to determine whether two mechanisms, one often at a higher level than the other, report the same status about some OS data structure. Such an approach is possible by combining the in-band and out-of-band delivery patterns and relying on the fact that a capable malicious entity will compromise the internal view-generating function or the guest OS itself, thus leading to an inconsistency between the two views created, which is then detected. Such an approach can also be described with our model:

\[
\begin{align*}
    f_\mu &: S_{int} \rightarrow V_{int} \\
    g_\mu &: S_{ext} \rightarrow V_{ext} \\
    a &: V_{int} \times V_{ext} \rightarrow P \\
    d &: P \rightarrow C
\end{align*}
\]

Using this formal description, it is clear which view-generating functions are used and that this is a combination of the in-band and out-of-band delivery patterns. This information
along with a thorough understanding of the the properties associated with the three patterns allow for a more structured analysis of a particular approach.

<table>
<thead>
<tr>
<th>Property</th>
<th>Delivery</th>
<th>Derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>in-band</td>
<td>out-of-band</td>
</tr>
<tr>
<td>HW portability</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Guest OS portability</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Binding</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Isolation from guest OS</td>
<td>—</td>
<td>✓</td>
</tr>
<tr>
<td>Inspection of suspended VM</td>
<td>—</td>
<td>✓</td>
</tr>
<tr>
<td>Full state visibility</td>
<td>—</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 1: Comparison of the properties of the different view-generation patterns.

We have summarized the properties of the different patterns in Table 1. When combining such patterns, it is important to carefully consider how the properties of these patterns will interact with each other. Deciding what approach is best for a given application is not as simple as combining all patterns in order to get the benefits of each. An advantage of one pattern will not necessarily negate the disadvantage of another. For example, a combination of a delivery pattern with the derivation pattern will not result in an approach that is guest portable simply because one of the patterns has such a property. In fact, in this case, the opposite is true: The disadvantage of one pattern negates the advantage of another.

There are additional cases in which the properties are independent, for example, a combination of a delivery and a derivation pattern, though this time with respect to the binding property. A combination of two such patterns can neither be classified as binding nor as non-binding. There is simply some information within the system that is the result of bound knowledge and some information within the system that is the result of unbound knowledge. How this information is to be used needs to be carefully considered.

Finally, some patterns are simply not appropriate for a given application. For example, using the derivation method alone to monitor changes in a file on a disk is a poor choice as it is impossible to derive the exact sector of a disk on which the appropriate portion of a file lies without knowledge of the file system structure in use.

5. APPROACHING A VMI-BASED IDS

Along with the formal model presented in Section 3, we have described the possibilities of VMI-based security applications in general. Our targeted security application for VMI is an intrusion detection system. As already mentioned in Section 2, building such an IDS faces three major challenges. Let us revisit these challenges with regard to intrusion detection and link them to our formal model:

1. The state information of a virtual machine needs to be interpreted. To accomplish this, the “semantic gap” as explained in Section 4 must be bridged. This is equivalent to constructing the view-generating functions $f_{\lambda, \mu}$ and $g_{\lambda, \mu}$ for a combination of architectures $\lambda$ and $\mu$.

2. The state information that is relevant for distinguishing between a compromised and an uncompromised system needs to be identified. In our model, this challenge corresponds to the aggregation function $a$, which builds and updates a meaningful profile from a sequence of consecutive views of a system run. In a practical application, it also involves view generation because the aggregation function is reliant on the information that the view-generating functions deliver.

3. The system state which is described in a profile needs to be classified as either “compromised” or “uncompromised”. It corresponds to the decision function $d$ of our model. As we will see in Section 5.3, this challenge can be well described as a machine learning problem.

In the remainder of this section, we will detail how a VMI-based IDS would ideally look like in theory and how we are approaching such an IDS in practice. We are currently implementing a VMI-based IDS using the Linux KVM hypervisor [10] on an AMD64 architecture as the host and a Linux-based guest OS. The overview of the ideal system can be seen in Figure 2.

Figure 2: Overview of the architecture of our VMI-based intrusion detection system. In this figure, $[\bullet]$ stands for “component implementing function $\bullet$”. The circled $p$ represents the profile that is stored within the IDS VM and updated in each step.

To minimize the risk of the VMM being compromised by a flaw in the IDS itself, the introspection is done in a separate VM, which we call the IDS VM. For this purpose, we extended KVM to allow the pseudo-physical memory of the inspected machine to be accessible as a special device node from within the IDS VM. Any introspection software is then run inside the IDS VM and analyses the memory of the monitored VM by reading from this special device node. The reconstruction of the file system of the stable storage of a KVM-based virtual machine is trivial as the disk image of the monitored VM can be attached to the IDS VM as an additional disk. This way the monitored disk can be directly mounted read-only in the IDS VM and then accessed as usual. Thus, the state information of the monitored guest
is made available to the IDS VM by the VMM, which is depicted in Figure 2 by the lower solid arrow.

Furthermore, we have implemented a hypercall and a signaling mechanism into KVM. The hypercalls enable the IDS VM to communicate with the hypervisor, e.g., to suspend the monitored VM for a consistent memory analysis. The signals provide a communication channel in the other direction, namely from the hypervisor to the IDS VM. This is used for informing the IDS VM about certain events asynchronously, e.g., when a specific memory region of the guest VM has been modified. These mechanisms are illustrated by the dashed arrows in Figure 2.

The reactive component is the controlling logic of the IDS. It coordinates the introspection and intrusion detection processes and triggers configurable actions based on the classification of the monitored VM states. Introspection of the VM is done in regular intervals. In order to do this, the reactive component may suspend the monitored VM via hypercalls to the hypervisor. The interval length can be adapted according to the required temporal granularity and the security level of the monitored system. Additionally, the KVM hypervisor may signal the reactive component if the monitored VM attempts to write into memory regions that normally do not change during regular system operations. For example, if the guest’s interrupt vector table is modified, the hypervisor signals this to the reactive component which may then perform an instant system check out of schedule.

5.1 Practical Interpretation of State Information

The interpretation of a binary string representing the system state results in a more abstract view of the state. In an ideal IDS, the view would be a reduction of exactly those parts of the system state that are relevant to intrusive behavior within the kernel and user-land processes. In addition, the view would be laid out in a uniform way which is independent of any randomness of the kernel’s execution (see Section 3.1).

It is clear that the creation of this ideal view is practically infeasible for highly complex systems such as modern computers. The selection of intrusion-indicating state information and stripping of randomness requires a full understanding of the whole kernel and all running applications. Even with the complete source code and technical documentation of the hardware and software at hand, this would require a tremendous amount of work.

Fortunately, a less complete view is sufficient for a practical IDS approach. As our focus is the detection of kernel-level attacks, we ignore all user-land processes and provide a view of the kernel-space only. In addition, we concentrate on extracting information from the state that has shown to be relevant for detecting intrusions, i.e., data which is actually modified by an intrusion. We will detail how we identify the relevance of information prior to extracting it in Section 5.2.

For our implementation, we use debugging and digital forensic methods to interpret the state information. We developed a tool that is able to reconstruct nearly any of the kernel-level data structures from a physical memory dump and output its contents. To accomplish this, the tool requires the debugging symbols (DS) and the System.map file of the running kernel. The former can be obtained by building the Linux kernel for the guest OS with debugging flags enabled, the latter is automatically generated during every kernel build. In terms of our model, the view-generation only makes use of the software architecture description µ, which in this case is µ = {DS, System.map}. With this information at hand, the tool reads the pseudo-physical memory of the guest OS through a special device node from within the IDS VM, as already said before, and makes the interpreted kernel-level data structures available to the aggregation function.

5.2 Experimental Identification of relevant State Information

We introduced the notion of a profile in Definition 5 abstractly as an aggregation of views. In other words, a profile may consist only of the last view as-is, but may also contain accumulated information from a series of consecutive views of a system run. In order to distinguish a compromised system state from an uncompromised state, the profile must contain the information that makes up this difference. This knowledge of which information is relevant must be available in advance because the aggregation function a is reliant on the information that fX,µ and gX,µ provide, and all three functions are constructed before a system is monitored.

An “ideal” VMI-based IDS would incorporate all knowledge about any available attack vector and system weakness and thus generate “perfect” profiles. Of course, this is not possible in practice. Even though there is some excellent literature available that covers many stealth techniques of computer malware [15,6], it does not answer the question how their presence looks like from a system state point of view. However, we know from the properties of VMI (see Section 2) that a VMI-based IDS can be provided with a complete and untainted view of the system state, thus being able to observe any system activity, whether benign or malicious. This property can be utilized to actually detect relevant state information.

As a consequence, we suggest an iterative and experimental approach for overcoming the challenge of building a valuable profile. One such iteration consists of a single experiment and is described in the following:

1. The state of a test system inside a VM is monitored while either a specific attack is driven or some normal system usage is simulated.
2. The state information that has changed is identified by comparing it to the initial state. A change set is generated from this comparison. This change set is mapped to the relevant kernel-level data structures or user-level processes.
3. The relevant kernel-level data structures are interpreted by the view-generating component2, the changes to user-level processes are ignored. If certain data is found that has changed but cannot be interpreted, the view-generating component must be revised to cover that gap.
4. At the point where all the state information that has changed can be interpreted, all parts which are crucial to system security are identified. This is a manual task and requires a sound knowledge of the kernel operations.

2In our implementation, this is our above mentioned view-generating tool.
5. The security-relevant parts of the state are added to the aggregation component so that this information is recorded in the profile in the future.

This procedure should be repeated with as many samples of benign and malicious activities as possible. Using this iterative approach, it can be verified that both the view-generating and aggregating components completely cover all information that might be relevant to each performed attack and thus generate very sophisticated profiles.

### 5.3 Classification of System States

The classification function $d$ bases its decision solely on the profile that it receives as input. In this context, a profile can be seen as one or more bit-strings containing the aggregated state information. How this information is actually represented and laid out is implementation dependent and irrelevant for the classification.

An ideal VMI-based IDS would always classify a given profile correctly as “compromised” or “uncompromised”. In reality, however, we do not have the complete knowledge and understanding of how a compromised system state can be distinguished from an uncompromised state by only looking at an isolated profile. Only a policy engine with a static rule set would be able to perform such a classification, but might be easily tricked if the attack is slightly modified.

Our ambition is to create a classifier that is able to learn and adapt to new attacks. Thus, we suggest the following approach.

From the experiments we do for identifying the relevant state information (see Section 5.2), we collect a data base of profiles describing compromised and uncompromised states. With this basis, we can formulate the classification problem as follows: Given two distinct sets of compromised profiles $P_c \subset P$ and uncompromised profiles $P_u \subset P$. Which class does an unknown profile $p \in P$ with $p \not\in P_c \cup P_u$ belong to? This problem statement is a well-known machine learning problem and has been a topic of research for decades.

For our IDS implementation, our plan is to compare different classification approaches for so-called supervised learning algorithms that produce general hypotheses from the training sets, e.g., Neural Networks, Support Vector Machines, and other kernel methods [2]. There already exists some work on machine learning for intrusion detection from several groups [11, 1, 14] which we will consider. The goal is to find the best suited algorithm or a combined approach for an accurate classification as well as to automate the process of finding the most significant features for classification in the training sets we generate.

Such machine learning approaches have several advantages over a static set of policies. First, they are able to generalize from their training sets to unknown elements. This makes them well-suited to classify states that have not been encountered in that form before. Second, most of these approaches do not make all-or-nothing decisions, but rather give a probability that an element belongs to a certain class. This allows the reactive component to induce fine-grained reactions to various threats, depending on their severity. Third, some of these algorithms support on-line learning, that is, classifications with a very high probability can be used to instantly update the classifier.

Even though machine learning approaches are more difficult to apply to a profile than matching it against a static set of rules, the advantages of these methods indicate that this effort will lead to better classifications and more flexible reactions.

### 6. RELATED WORK

Since the work done by Garfinkel and Rosenblum [4], there has been quite a bit of work taking similar delivery approaches, all of which can be described with our model and the patterns we identified. The VIX tool suite [5] is an implementation of the out-of-band delivery pattern for forensic analysis. Basing a forensic tool on this pattern is the optimal choice, as this pattern provides many properties that work to the benefit of this particular application. For example, such an approach will provide a consistent and complete view without interfering with the state of the guest being investigated. In addition, such a host can be immediately suspended without loss of information until a thorough investigation is complete. The only noteworthy drawback of this particular pattern is its non-binding nature. However, in the forensics field, such a tool is often used as one of many and an expert analyst can always forgo the use of such a tool for another method (e.g., manual investigation) if no results are found.

Jiang, Wang, and Xu [7] present another example of an implementation of the out-of-band delivery pattern, though they focus on malware detection through hidden process detection and performing virus scans on the guest OS’s hard disk from the VMM. Again, this approach benefits from many properties of the out-of-band delivery pattern, such as isolation and the ability to monitor a suspended system. The authors go on to combine the out-of-band and in-band delivery patterns to demonstrate the usefulness of their approach. They allow their software to run within the VMM (out-of-band delivery), while some other software runs within the guest (in-band delivery), in order to show the effectiveness of their approach. Being an implementation of the out-of-band delivery pattern, this approach has a non-binding nature. Jiang, Wang, and Xu refer to an attack on the non-binding nature of their approach as a “guest view subversion attack”.

In order to move away from an approach with the non-binding property, it is necessary to consider derivation approaches. Litty et al. [13] present a system they call Mani- tou. It uses the paging mechanisms of IA-32 processors from Intel and AMD to perform integrity checks on code pages before execution. They further this work in [12] to create a system called Patagonix for reporting running processes and maintaining the integrity of code pages. This work uses knowledge of the hardware architecture, specifically the paging mechanism, to perform its intended tasks. Such an approach is clearly an implementation of the derivation pattern and is therefore bound to this hardware architecture.

Jones et al. [8, 9] take a slightly different approach, but also present a system that implements the derivation pattern. Their system makes use of the paging mechanism and the memory management unit (MMU) in x86 and SPARC hardware architectures to identify running processes. This is another great example of a derivation approach.

The two previously discussed approaches are the only examples of systems that implement the derivative pattern to our knowledge. They show that such an approach is feasible, but are, for now, somewhat constrained in what they can do. Developing new derivation methods takes a great deal of understanding of hardware architectures and there
simply are limitations to such an approach. One cannot derive the meaning of every bit of the state based solely on knowledge of the hardware architecture. So, while a derivation approach does have the attractive property of being binding, the use of such an approach will always be very specific and may need to be combined with approaches implementing other patterns for a complete solution for the particular application. Further exploration in this direction is an interesting proposition.

7. CONCLUSION

In this paper, we have presented a universal formal model for virtual machine introspection. It is universal in the sense that it can be applied to all security applications of VMI we are aware of. This model is meant as a means of reasoning about and comparing possible VMI solutions for various applications. It is in no way meant as a tool for formal verification. This model has shown its usefulness throughout our research. To demonstrate this usefulness, we have classified all possible view-generating approaches into three patterns, namely in-band and out-of-band delivery, and derivation. Furthermore, we presented a thorough discussion of each pattern’s properties as well as their advantages and disadvantages for view generation.

In addition, we identified and discussed the three major challenges of VMI-based security applications, namely, interpreting the state information, identifying relevant state information, and classification of states. Since we are considering VMI from an intrusion detection perspective, we reasoned about the theoretical limitations of a VMI-based IDS as well as shared our thoughts about the constraints in a practical implementation of such a system.

As we have found our model an invaluable tool for such discussions, we urge readers to make use of the formalisms presented in this paper to help better understand all the intricacies of building VMI-based security applications. The model and the associated patterns are able to represent all VMI approaches in a consistent way, making them comparable and allowing one to reason about their properties and appropriateness for a given application.

8. REFERENCES


