Abstract—The commercial success of Cloud Computing and recent developments in Grid Computing have brought platform virtualization technology into the field of high performance computing. Virtualization offers both more flexibility and security through custom user images and user isolation. In this paper, we present an approach for combined malware detection and kernel rootkit prevention in virtualized Cloud Computing environments. All running binaries in a virtual instance are intercepted and submitted to one or more analysis engines. Besides a complete check against a signature database, live introspection of all system calls is performed to detect yet unknown exploits or malware. Furthermore, to prevent that an intruder retains persistent control over a running instance after a successful compromise, an in-kernel rootkit prevention approach is proposed. Only authorized and thus trusted kernel modules are allowed to be loaded during runtime; loading of unauthorized modules is no longer possible. Finally, the performance of the presented solutions is evaluated.

I. INTRODUCTION

External and internal intrusions are the most serious threats in computer systems connected to a network. Attackers exploit software bugs in core components on a target system to gain superuser privileges, allowing the attacker to take control of the attacked system. The rise of Cloud Computing aggravates the stated problem. Cloud Computing refers to both the on-demand provisioning of hardware resources in the data centers of public providers such as with Amazon’s Web Services, and the applications delivered as services over the Internet, such as with Google’s AppEngine. Offering access to remote compute resources is often referred as Infrastructure as a Service (IaaS). The resources provided as IaaS are platform virtualized environments, i.e., customers have access to their own virtual appliances running on shared physical resources.

To retain the control of the attacked system persistently, the intruders typically install malware in order to recover full control after reboot. The target system in this case is the virtual machine offered by the Cloud Computing provider. This kind of attack is commonly discussed as a strong intrusion attack, while temporary attacks between two operating system startups are referred as weak intrusion attacks [1]. The software toolkits that are installed within a strong intrusion attack are commonly called rootkits. Usually, weak intrusion attacks are used to place a rootkit on the attacked system.

These potential threats create the need for a new malware detection system. Providers need ways to ensure the security of their infrastructure and the systems of their customers. Having a flexible Cloud infrastructure also opens new possibilities to scale up and distribute malware detection software among several systems. Most end-host security solutions have a major, negative performance impact on the computer caused by huge signature-sets or complex detection algorithms. Cloud Computing can be beneficial here to decrease the slowdown and offload it to dedicated machines.

In this paper, we present an approach that deals with malware detection and kernel rootkit prevention. While the former deals with detecting malware traces during runtime in a safe and non-intrusive manner, the latter prevents rootkits from being installed in the operating system kernel. A Cloud-based intrusion detection system to recognize running malware is designed to run on virtual machine instances with a backend Cloud to distribute malware scanning operations between several backends. A flexible framework for a distributed security solution with a minimal overall resource footprint on the end host is presented. To detect well-known as well as yet unknown malware, a traditional signature check is performed and the prerequisites for a live system-call tracer are presented. Furthermore, the solution introduces an integrity check of authorized kernel modules to prevent rootkit installations via corrupted kernel modules. For this purpose, the operating system kernel is modified to load only previously cryptographically authorized kernel modules.

The paper is organized as follows. Section II states the problems involved in virtual machine distribution. Section III presents the design of the proposed solution. Section IV discusses implementation issues. Section V presents results of several performance measurements. Section VI discusses related work. Section VII concludes the paper and outlines areas for future work.

II. PROBLEM STATEMENT

A convenient way for an attacker to gain control over a compromised system is a rootkit. There are various types of rootkits available, e.g., application level rootkits that replace the original binaries with a fake binary containing a trojan horse or library rootkits that replace valid library functions with malicious ones. The focus of our work is the kernel rootkit. It replaces/adds functions or device drivers in the kernel space of an operating systems. Kernel modules in general enable upgrades to specified parts of a kernel to strengthen modularity of the operating system. There are two
classes of kernel modules: permanent kernel modules, which are loaded at boot time and cannot be removed once they are running, and loadable kernel modules, which can be loaded and unloaded by the system at run time. Many kernel rootkits are designed as loadable modules or device drivers, since this is the easiest way to add new functionality to the core system. Thus, monitoring the loading process of kernel modules is indispensable to ensure that no malicious modules are loaded.

There are various ways to disable dynamic kernel module loading:

- In Linux, it is possible to disable kernel module loading completely. While configuring a kernel, the administrator can set the `MODULES` option to `NO` and thus disables the complete kernel loading and processing mechanism. While this completely prevents kernel rootkits from loading, it also affects all legal modules.

- The technique of multiple secure levels is used in various BSD derived Unix operating systems. Any super user is able to increase the secure level. On the other hand, the only way to lower the secure level is via the init-process, a prototype user process that is only loaded during system startup, so the system has to be restarted. For example, FreeBSD [10], a widely used Unix branch, runs with four different levels of security.

Thus, it is possible to disable dynamic module loading either by disabling modules or via a higher secure level. In this case, one has to take the good with the bad: On the one hand, this avoids critical actions such as arbitrary changes of kernel memory through user programs (which, in fact, is performed by loading a kernel module). On the other hand, the concepts are very restrictive and forces users to compile and install the whole kernel instead of linking a single file. This step makes a reboot of the modified system necessary and interrupts running applications. Actually, for several applications (e.g. all mission critical applications), this is not a suitable solution.

While the previously stated problem applies to a greater extent to infrastructural machines, such as critical servers (e.g. DNS, DHCP), a Cloud provider should also be interested in keeping the VMs of its customers safe. Most Cloud vendors provide VMs with full root access, meaning that a user can mostly do whatever (s)he wants, including destroying the whole machine. Since Cloud Computing is about pay-as-you-go, this should not harm the vendor. Nevertheless, if a user (intentionally or not) executes malware, this could also affect the provider, e.g. a spam malware could abuse the outgoing bandwidth to send mass-spam mails. Thus, while granting root permissions to its customers, a provider still should be able to inspect the applications running inside its customers’ VMs.

Furthermore, (s)he should be able to take countermeasures if (s)he detects a security violation, such as running malware binaries. In the following section, we will present a Cloud based host intrusion detection system with a minimal resource footprint as well as hidden from the malware itself in the operating system kernel.

III. Design

In the sequel, the proposed solution to the problems stated above is presented. Our proposal is based on the standard assumptions made in most other virtualization security architectures [3], [5]. The hypervisor is part of the trusted computing base (TSB). Since we focus on infrastructural security, we do not deal with attacks against the Virtual Machine Monitor.

A. Malware Detection

Contrary to a classic anti-virus setup, a Cloud specific design of a malware detection engine should run in a distributed manner and display some special requirements to ensure the security of the service provider’s infrastructure as well as the customer’s security. The communication paths and different software modules of our proposed design are shown in Figure 1. Any program run by the user (1) is executed in a virtual machine in the Cloud. The kernel of this machine then passes all relevant information to a KernelAgent (2). The KernelAgent gathers all information by the virtual machines running on the Cloud resource and then relays them to the ScanProxy (3). The ScanProxy provides a front-end to the Cloud security analyzer services. At this stage, the proxy has to distribute the information to the different services, such as classic anti-virus software or behavior-based analysis solutions (4).

![Fig. 1. Malware scanner architecture](image-url)

The kernel module is the primary sensor sitting directly in the running virtualized kernel of the guest machine. To avoid any security issues through the kernel module, it has very limited functionality. Its main task is to function as a logging relay and to submit any interesting activities to the KernelAgent for further processing. Valuable information include process creation or termination, system calls by these processes and the system call parameters as well as any binary files getting executed. This approach makes it very easy for the Cloud provider to maintain the system. The only component that has to be changed is the kernel. Thereafter, all operating system (OS) images that are provided by the
customers can be booted using the modified kernel. Contrary to classic anti-virus solutions, no installation within the OS image is necessary, which means the additional security provided by the OS is completely transparent to the customer. Moreover, the customer has full control over his or her OS image. No matter what the customer does with the image, (s)he cannot break or deactivate the malware detection system.

The kernel modules should intercept any executable before it is running and submit it to its host agent. This is the way classic anti-virus hooks grab an executable before loading it into their scan engine. They check every executable through static analysis. Applying static binary analysis might not always be the best way to ensure security, especially when confronted with unknown, new malware. Nevertheless, it still should be part of any malware detection solution. Using this approach, it is easy to take advantage of all the existing anti-virus software. A requirement for any executable analysis is the binary image of the file itself, and for identification purposes, the filename must also be transmitted.

1) Process Life Cycle, System Calls: Monitoring a process with respect to its system calls throughout its lifecycle can be a valuable source of information when looking for common patterns in malware behavior. By relaying this information live, not only encrypted executable images and obfuscation, but also in-memory injected malware through an exploit can be analyzed. System calls make it easy to spot specific file accesses or socket operations, such as transmitting data back to an attacker. The relevant information includes the system call, its parameters, return values and the program that made the call.

2) KernelAgent: This part collects all the data from the VM kernels running on the host system. This information should then be relayed to the ScanProxy. Since there is no other logic involved in this piece of software other than the configuration of what has to be sent to whom, there is almost no need to touch an installed system. To increase the performance, caching of messages and later on responses is implemented. This is especially interesting for classical executable image analysis. While starting-up several virtual machines, the same executable is run several times. These often called binaries include e.g. system services. Submitting and analyzing the executable at every initiation/run costs CPU time and also increases network traffic. This can slow down the start-up time in a feedback based intrusion prevention system significantly.

Since both groups of information (binary and system call related) have different requirements, splitting up the KernelAgent into two separate servers makes sense. One is a UDP-based system call forwarder, the other one should receive binaries and forward them. The binary executable relay must not save any executables to the hard disk. Otherwise, there is a chance of an infection happening on the host system in case of malware.

3) ScanProxy: This component gathers all available information from the hosts and distributes it among the registered scan engines. For each incoming packet containing a system call, one or more receiving scan engines can be used. The proxy then forwards the packet to the registered receivers. It could also act as a global log and caching proxy for the complete Cloud. Every new scan engine being a system call analyzer or a classical anti-virus scanner can be enlisted here once or even several times for redundancy purposes. The proxy does not need to have much more logic than the above to keep the system as easy to manage and immunized as much as possible. More code complexity means more space for failure through attacks.

4) Scan Engine and Executable Analysis: Considering the previously described framework, several possible scanning backends can be implemented. They can generally be categorized as process behavior based or executable binary based, such as a classical anti-virus solution, e.g. ClamAV [2]. Every incoming executable has to be placed in a separate container on the hard disk and then has to be analyzed. The received binaries must not be executed, otherwise the security of the scanning computer can be compromised in case of an infection. By registering several different anti-virus scanners with wrappers, an increased level of security can be achieved. This helps to minimize the vulnerability window that exists between the discovery of a new malware and the release of the signatures by the anti-virus vendors for their products.

To process events such as systems calls, a backend like the software of Wagener et al. [13] can be used with minor modifications. The underlying concept of their approach is that even new malware shares common behavioral similarities to already existing malware. By finding these similarities in behavior graphs, even yet unknown malware can be automatically detected. While Wagener et al. perform system call analysis ahead-of-time in a secure execution environment, modifications should easily be possible to enable on-the-fly detection.

B. Kernel Rootkit Prevention

Instead of disabling kernel module loading completely, we focus on the BSD secure level concept. It allows module loading before raising the secure level. The following subsection describes the process of kernel rootkit prevention by loading authorized kernel modules only. We distinguish between two modes, describing the state of the secure level. If the secure level is lower or equal than 0 (which is the default for single user mode), it is called insecure mode; if the secure level is set to 1 or higher (kernel memory is read-only, file system might be read only), it is called secure mode. Furthermore, adding a module to the internal list is called mark/unmark as authorized.

To prevent kernel rootkits, we distinguish between safe and unsafe kernel modules. In secure mode, it is only possible to (un-)load authorized kernel modules. It is not possible to load other modules, especially any kind of malware. All authorized modules are kept in a list that resides in read-only kernel memory. The latter is needed to prevent that an attacker could simply modify the list to mask a rootkit as an authorized module. Each list entry contains the following information:

- A human-readable description of the kernel module
- A unique cryptographic hash of the kernel module
- Some internal kernel structures to indicate whether the kernel module is currently loaded
The implementation uses a generated SHA256 hash to provide a unique key for each module. To authorize kernel modules, a userland program has been developed to add or remove kernel modules to the mentioned list through a system call. This system call refuses execution if it is called without root privileges. Optionally, all dependent modules could be added as well. Any operations on the list can only be made while the system is in insecure mode. A convenient moment would be the initial system setup before it is actually connected to an external network. While the system is running in insecure mode, the userland program is able to mark/unmark modules as authorized. The list, where the marks are stored, uses transient storage, i.e., the list is initially empty at system startup time.

Figure 2 shows the possible modifications of a list entry. By default, a module is not loaded and not authorized. In insecure mode, a user can mark a module as authorized and thus is able to load it later when the system is in secure mode. All kernel modules that are loaded during the boot process (e.g., the ACPI subsystem or device drivers) are not authorized. Consequently, they have to be authorized before the system is switched to secure mode. Otherwise, they would work as expected, but unloading would not be possible (which might not be necessary, especially if it is a core component). Once the system is in secure mode, only authorized kernel modules can be loaded.

The main features of this process are encapsulated in the dynamic module loading process to check whether a module is marked as authorized or not. To authorize a module, an authorization function has to open the module file, hash its content and search for matching hashes in the list. If the authorized-flag of the corresponding list entry is set, the module is allowed to be loaded. The unloading process is handled by another function that checks if the module is already loaded. Consequently, we do not have to hash the module again. Every loaded module is equipped with a unique pointer, representing the module. This pointer is used to find the correct module in the list and to decide whether to unload or not. Finally, unloaded modules must be marked as not loaded in the list.

IV. IMPLEMENTATION

Like the operating system kernel (which in our case is the DragonFly BSD kernel, version 2.5.0), the kernel part of the malware detection module and all parts of kernel rootkit prevention and have been written in ANSI C.

A. Malware Detection

To tap into the relevant parts of the kernel, some static hooks are installed. These hooks redirect or copy valuable information from kernel functions such as execve to an extra function that passes this information on to the KernelAgent.

1) Process Related Information: Getting all process related information requires the addition of several hooks to the VM kernel. A hook is installed in the function that adds new processes to the kernel’s process list and assigns a new PID to them. The list is a linked-list used to keep a global list of all running processes. Another hook that is called at the end of a process lifetime works in a similar fashion. This routine is called by the kernel’s exit function to remove a process from the global list of running processes and add it to the list of dead processes. This list is an in-kernel linked-list containing all processes in the ZOMBIE state. This means that they are about to be removed from memory and are done executing.

The system call hook is called from within the VM’s syscall2 function. It is executed immediately after the processing of the real system call. Getting called after the execution of the system call has the advantage that some parameters that are passed on empty to the kernel and are filled during execution can get their content inspected. This is, for example, the case with the open system call that has a buffer as its parameter for reading bytes from a file descriptor.

The challenging part here is getting the parameters. They are passed to the system call function without providing any type-information, except a memory reference. For the kernel, there is no need to know this type-information, since the corresponding system call knows what type its parameters should have. As part of our approach, an extra file holds a list of all system calls and their parameter types. Additionally, the error code as returned by the actual system call is provided for analysis purposes. This has the advantage that the data flow can be recorded, such as the returned file descriptor from an open call and later on any read system calls to this file descriptor.

2) Executable Loader: The binary loader hook is placed in the kern_execve function of the VM, which is the actual place of execution and not the system calls’ first entry point, sys_execve. To avoid unnecessary calls to the logging hook, it is only called after exec_check_permissions has successfully returned. After this call, it is certain that the executable is valid and has the appropriate permissions. The logging takes place before the first page of the executable gets mapped into the memory and is executed. In a feedback-based Intrusion detection system it would still be possible to stop the execution
at this stage, should the binary be infected with malware. The whole binary is then submitted to the KernelAgent using TCP.

3) Communication from the VM to the Host System: To keep the protocol overhead as small as possible and be as responsive as necessary, a simple protocol is implemented by using UDP in the kernel. Approaches based on TCP would have brought up some additional delays, which is a problem when monitoring realtime events such as system calls.

4) KernelAgent: The KernelAgent’s main task is to collect the data from all virtual kernels running on the machine and forward it to its ScanProxy. This part is implemented using the Python scripting language. Whenever a new packet is received, a background thread is started to process the received packet. This implies that it is parsed and then the whole packet is sent forward to the configured ScanProxy.

5) ScanProxy: This software module is similar to the KernelAgent on the receiving part and is also written in Python. Instead of one configured receiver for relaying, like in the KernelAgent, there is a list of receivers. This list can be configured for each entry to relay only specific types of traffic (e.g. only NEWPROC, ENDPROC and SYSCALL) or any traffic for a catchall or logging daemon. Due to this fine grained configurability, the incoming packets must be inspected and checked against the list of receivers to ensure that every receiver obtains only events that have been subscribed for.

For scanning executable files with an anti-virus software such as ClamAV, a TCP variant of the ScanProxy has also been written. Just as within the UDP ScanProxy, a list of receivers/beckends can be configured. All incoming binaries are relayed to them. The ScanProxy only keeps the executable’s data in memory, nothing gets written to the hard disk. This backend checks incoming binary files with ClamAV for known viruses. Incoming files are received over TCP connections to ensure that the received binaries are complete and in-order. As in the KernelAgent and the ScanProxy the name of the executable is also submitted. Every received binary is saved in a temporary quarantine folder, where it is scanned. After scanning is done, the file gets deleted to ensure security of the backend system.

B. Kernel Rootkit Prevention

All main communication between userland tools and kernel is handled by a newly introduced system call, e.g. add a new module to the internal list is done by sending the required information via the defined interface. As mentioned before, the internal list has to hold any information about a module. For example, a module can just be authorized, not loaded, or it can be loaded but not authorized. Therefore, our list contains one entry per module. The state is held in flags or implicitly by pointers being not empty.

Every generated list entry holds a unique key. In our implementation, we use the generated SHA256 hash to provide a unique key for each module. The longer the resulting hash value is, the more secure is the corresponding algorithm regarding brute force attacks. Thus, SHA-256 is a good tradeoff in terms of security as well as memory usage and performance.

The identifier is used to hold the human readable description of that entry. A linker file pointer points to the corresponding linker file kernel structure. This is a convenient way to map a loaded module to the generated hash without doing changes inside the existing kernel structures. To prevent the list from being changed while system is in secure state, the functions responsible for mark-and-authorize a module are not callable in that case. After switching to secure mode, only authorized modules can be loaded or unloaded - there is no way to authorize kernel modules retroactively.

Usually a userland tool is used to load modules during runtime. This utility directly uses the system call kern_kidload, which basically implements dynamic module loading. After a basic permission check, the main module loading, depending on the binary format, is performed. Those formats are compiled into the kernel and cannot be changed dynamically. The common format is the Executable and Linking Format (ELF). Each of the functions above verifies the secure level at first and interrupts loading immediately if the system runs in secure mode. Figure 3 shows a flowchart of the module loading process (changes drawn in dashed lines).

![Fig. 3. Module loading activity](image-url)
the virtual node is opened without errors, the module is read. Otherwise, the function aborts with an error message. Since the read bytes can be added to the hash algorithm successively, the memory usage by reading data piecewise using the same buffer each time is reduced. This data can be used to generate the hash key. If the internal list contains the generated hash key, the module is marked as to be loaded, otherwise it is not and the appropriate permission denied error is returned.

Authorizing a module within the unloading process is less complex, because the used data structures by the unload process contain a file pointer that is also registered in the internal list if the module is loaded. If the module is authorized, it is unloaded. Otherwise, unloading is not permitted.

The diagram clearly shows that the host operating system and the appropriate permission denied error is returned.

Authorizing a module within the unloading process is less complex, because the used data structures by the unload process contain a file pointer that is also registered in the internal list if the module is loaded. If the module is authorized, it is unloaded. Otherwise, unloading is not permitted.

The upper bars, named sys, indicate the time spent executing system calls on behalf of the executed program. The lower bars, named user, represent the time spent doing calculations, iterations or generally spoken actions in userland. The diagram clearly shows that the host operating system easily outperforms the virtualized kernels. Even though the time spent executing system calls is nearly identical between host and the VM kernel, the time spent in userland is much more compared to the time when running on the host directly. Enabling the tracer functionality of the VM kernel doubles the time spent in the kernel, but the userland portion stays constant. Since the in-kernel time for system calls is so low compared to the total execution time (0.06 seconds), this impact on the performance can be neglected.

Figure 4 (b) shows the measured time needed to intercept a 8 KB binary in a running VM with the KernelAgent, transfer it over the network and scan it with the ClamAV engine. We conducted over 350 trials to get a robust mean, which is 0.5 seconds. The oscillation of the graph is due to the fact that we could only measure with a wall clock, which in this case works with Unix timestamps (seconds since the epoch). The measured overhead of 0.5 seconds before the actual execution starts is negligible in the described Cloud environment, since most jobs will be long running computational jobs. Furthermore, the use of caching techniques will even reduce the overhead, as every (unchanged) binary is only scanned once. Figure 4 (c) shows the times needed to transfer various binaries of different sizes over the developed middleware between KernelAgent and the ScanProxy. We conducted multiple measurements with different binary sizes representing different types of malware (the average file size of the standard system binaries is about 1.2 MB). For binary 1 (58 KB), the average time is 0.001 seconds, for binary 2 (685 KB), the average time is 0.06 seconds, for binary 3 (1.2 MB), the average time is 0.1 seconds, and for the biggest binary (2.4 MB), the average time is 0.2 seconds. Thus, the transfer time increases with the size of the binary.

B. Kernel Rootkit Prevention

To measure the module loading overhead, we wrote a script that cascades module (un-)loading. Since the main overhead is due to hashing the modules, we had to choose a proper average module size to get realistic results. By examining the standard kernel directory, we evaluated 64 kByte to be the average size of the kernel modules. In order to be a bit ahead, we created 100 KB sized kernel modules for testing. To be able to measure the correct time overhead, we loaded modules between 250 and 2000 times.

As shown in Figure 5 (a), we have an overhead in every measurement. The module hashing causes the overhead during every module load. Loading modules either in secure or insecure mode (with enabled kernel protection) takes more time than loading modules in the default mode (no protection and a stock kernel). This is due to the fact that the kernel rootkit prevention technique has to iterate over the internal list to validate a module. Thus, there is an additional linear effort. Nevertheless, module loading is not a time critical job and the average number of loaded modules should be much lower than in our tests. In the case of 250 loaded modules in the generic kernel, a module needs 0.016 seconds on the average to get loaded. In a kernel with rootkit prevention it takes 0.02 seconds on the average. This is more than 1.25 times longer, but still is not a large delay. If the system is running in secure mode, loading a module will consume more time, because there is one additional list iteration involved in the loading process. Generic kernels are not even able to load modules during secure mode. The measured overhead for module unloading is shown in Figure 5 (b). Contrary to the loading process, the unloading process is less time consuming. There is no noteworthy time difference regardless which kernel mode is used.

VI. RELATED WORK

Kroah-Hartman [8] has proposed to sign executables with a fingerprint. It is stored in an additional section of the commonly used executable linkage format (ELF). Furthermore, the technique of asymmetric cryptography is used to protect the fingerprint from malware modifications: A private key is used to encrypt the fingerprint stored in the ELF section, while the kernel linker decrypts the signature in order to compare it with the signature of the loaded file. A general problem is the kernel-level implementation of an asymmetric cryptography algorithm. There is no such implementation in most current
operating systems. This is the reason why we have chosen the symmetric SHA256 hash algorithm.

A similar way of implementing a kernel rootkit prevention technique is used by Catuogno et al. [1]. They implemented a verification mechanism based on encrypted signatures stored in an additional section of an executable as well. In contrast to Kroah-Hartman, they did not address dynamically loadable kernel modules but executables in general. This is why they assumed that the support of dynamically loadable kernel modules should be disabled. We think that this is not an appropriate assumption for the security of most applications.

In the NetBSD operating system, the Veriexec (verified execution) kernel subsystem allows users to monitor files and to prevent their removal, read/write access or execution if necessary [12]. It implements four levels of strictness: A learning mode for configuration matters, intrusion detection and intrusion prevention mode, as well as a lockdown mode. Contrary to Veriexec, the proposed solution is specialized to protect the kernel from modifications by dynamically loaded modules. We use a comparable database and fingerprints, but in contrary to the NetBSD kernel, we do not use the obsolete lkm (loadable kernel modules) architecture.

King et al. [7] have classified three kinds of malicious services supported by virtual machine based rootkits (VMBR): Services not interacting with the target system (spam relays, DDoS zombies, phishing web servers), services observing data or events (keystrokes, network packets) using virtual machine introspection and services deliberately modifying the execution of the target system. They successfully implemented all of these types combined with a countermeasure against the redpill virtual machine detection mechanism through emulating an instruction, which is used to determine a difference between a real and a virtualized processor’s interrupt descriptor table. Our intrusion detection approach cannot defend an attacked system once a VMBR is installed, nevertheless the proposed secure level mechanism is powerful enough to protect an endangered system from a VMBR installation by locking e.g. shutdown scripts used for Subvirt installation by King et al.

Garfinkel and Rosenblum [4] have described a virtual machine introspection based on an architecture that leverages the isolation, inspection and interposition properties of VMMs. Virtual machine introspection (VMI) describes a family of techniques that enables a VM service to understand and modify states and events within the guest. Beside this passive monitoring technique, active monitoring of virtual machine-based IDSes has been implemented as well [6]. Although they are facing the gap between the VMM’s view of data/events and the guest software’s view (which is called semantic gap), their modifications of the guest operating systems are detectable.

CloudAV [11] is a software stack developed by Oberheide et al. It is meant to counter the problems single anti-virus solutions face nowadays with the increase of different malware and new exploit techniques. Instead of having just one AV solution per host, CloudAV uses multiple, heterogeneous de-
tection engines. This approach is called ‘N-version protection’.

The Automatic Malware Signature Discovery System (AMSDS) [14] has been developed by Yan and Wu. The fact that increasing numbers of zero-day malware take more and more time to analyze and the need to write signatures for them indicates that it is necessary to provide automatic signature generators. Moreover, the increasing size of signature databases and analysis techniques increase the processor and memory footprint on computers with installed anti-virus solutions. This can be countered by anti-virus software as a Cloud service, putting the workload of analysis and signature maintenance on dedicated machines. AMSDS has a small detection engine with a reduced signature set. This set of signatures can match a great share of malicious software through special treatment and preprocessing of the binary. Only if the much smaller AMSDS signatures cannot detect a suspicious file, it is send to the Cloud anti-virus service for scanning with traditional anti-virus solutions. The automatic signature generation is very effective and space saving compared to classic signature generation. But these signatures can only detect binary executables loaded either from disk or network. An already running binary such as a service infected through an exploit is not covered by this approach.

Laureano et al. [9] have implemented some kernel introspection mechanisms into User-Mode-Linux. The authors gather information about the running system by inspecting the flow of the system calls made. Their IDS runs in two different modes: a mode for learning the regular behavior of a system and a so called monitor mode where anything unusual generates an alarm and suspicious processes are denied access to specific system calls. A similar system could easily be implemented within our framework. Furthermore, in our approach access to system call parameters is granted, enabling a more fine grained behavior analysis, while their approach just reports the system calls. Ignoring the parameters might lead to significantly more false alarms, since it can make a huge difference whether an open system call accesses a password file or just a new temporary file.

VII. CONCLUSIONS

In this paper, we have presented an approach for combined malware detection and rootkit prevention in Cloud Computing environments. All running binaries are intercepted by a small, in-kernel agent and submitted to one or more backend units where the actual classification process happens. Furthermore, live-scanning of all binary system calls is performed to detect yet unknown exploits or malware. Due to the in-kernel nature of the agent, it is completely transparent to the user as well as to malicious binaries trying to detect any countermeasures. The distributed architecture allows a good utilization of existing Cloud resources and the connection of different analysis engines.

While the detection rate of malware and anti-virus scanners has steadily improved within the last years, its still not a fool-proof solution against recent exploits like zero-day exploits. Many successful attacks lead to the installation of a kernel rootkit to gain permanent control over the target machine including the possibility to get access at later times and misuse the machines as an attack platform. Consequently, the proposed solution is a modification of the in-kernel loading process. Only authorized and thus trusted kernel modules are allowed to load during runtime. Loading of unauthorized modules is no longer possible.

There are several issues of future work. For example, the malware detection engine currently implemented provides a solid foundation for a flexible Cloud specific anti-malware solution. In a second step, it would be desirable to change the software stack from just a detection engine to a bidirectional intrusion response engine capable of isolating and terminating malware binaries in real-time. For the rootkit prevention solution it would be desirable to bring asymmetric cryptography into the various operating system kernels to be able to use signatures instead of cryptographic hashes. Finally, the possibility to manage the in-kernel black- and whitelisters (or a central signing key) could be realized by a central instance in the Cloud Computing environment. For this purpose, various parts of proposed infrastructure could be reused to achieve this goal.

VIII. ACKNOWLEDGEMENTS

This work is partly supported by the German Ministry of Education and Research (BMBF) (HPC Initiative).

REFERENCES