Using Hardware-Assisted Virtualization to Protect Application Address Space Inside Untrusted Environment

Denis Silakov
Institute for System Programming
at the Russian Academy of Sciences
Moscow, Russian Federation
Email: silakov@ispras.ru

***Abstract*—In this paper we present a virtualization-based approach of protecting execution of trusted applications inside potentially compromised operating system. In out approach, we do not isolate application from other processes in any way; instead, we use hypervisor to control processes inside OS and to prevent undesired actions with application resources. The only requirement for our technique to work is presence of hardware support for virtualization; no modifications in application or OS are required.**

***Index Terms*—Virtual Machine Monitor, Hypervisor, Security, Protection**

I. Introduction

In modern software world, an operating system is a key component responsible for many security aspects of application execution process. In particular, it should provide possibilities to manage access permissions of application files and other resources, guarantee isolation of application address space in memory and so on.

However, many widespread operating systems (such as Linux or Windows) are known to be subjected to vulnerabilities which can be used by malicious code to compromise the whole system or particular application. As operating systems evolves, vulnerabilities are detected and fixed. But at the same time a lot of new features are added which potentially introduce new vulnerabilities. Size of code which is executed with highest privileges in modern OS is large. In particular, many popular systems are based on monolithic kernel where every device driver is a part of the kernel (that is, works in the same address space with other kernel parts and other drivers). It is common for drivers to contain issues. Vulnerability research performed in 2005 has shown that device drivers were responsible for about 85% of failures in Windows XP [1]; similar statistics was reported for Linux [2]. It is very likely that the situation will not change in the near future, since size of drivers grows faster than size of any other part of the kernel [3].

Microkernel-based operating systems are claimed to be more secure due to the fact that the size of code executed in privileged mode is very small [4]. However, in such systems interaction between micro kernel and drivers which becomes quite expensive. If used on a desktop machine with lots of peripheral devices, such systems often demonstrate worse performance. In addition, there are a lot of applications developed for widespread OSes with monolithic kernels. It would be very expensive to port all these programs to a system with completely new architecture. As a result, nowadays microkernel-based systems are primarily used either in highly tailored areas (e.g., QNX for embedded real-time systems) or for educational purposes (e.g., Minix).

Thus, there is a need for application protection techniques that will not require modifications of existing operating systems or applications, but at the same time will provide more reliable and secure services than traditional approaches.

One of the possible techniques is to use hardware-assisted virtualization. As implemented in modern Intel and AMD processors, it allows to launch a program (called *hypervisor*) that has full control over hardware and runs with higher privileges than OS. Normally, hypervisor is responsible for virtualization (e.g., creating and managing virtual machines), but its functionality can be enhanced. In particular, it can provide some security services. Hypervisor is usually much more smaller than OS (for example, most hypervisors do not have a large set of device drivers). As a result, hypervisor potentially contains less vulnerabilities and usually considered to be more secure than commodity operating systems. In this paper, we suggest an approach for protecting application address space using hypervisor.

The remainder of the paper is structured as follows: Section 2 observes existing virtualization-based approaches to protection of application resources. Section 3 describes general architecture of our protection system and specific aspects of protecting address space of applications of different kinds. Section 4 describes implementation of our approach and present performance measurement results. Finally, Section 5 summarizes the main ideas.

II. Hypervisor-Based Protection Systems

The idea of using hypervisor for different security tasks has got much attention after Intel and AMD introduced their first implementations of hardware-assisted virtualization in

years 2005-2006. Many approaches requires modifications of applications, OS (e.g., [5] or [6]) or even hardware ([7], [8]). Though some of these approaches seem to be quite effective, their usage is rather limited.

A promising approach is Overshadow technology of memory protection suggested by researches from Stanford and Princeton Universities, MIT and VMware, Inc ([9]). It does not require modifications of OS or applications. Instead, it encrypts process memory area of working processes. If OS or other program try to access process memory, they only see encrypted data. For trusted process itself, a "normal" memory view is provided. Similar approach based on dynamic encryption of application memory is presented in [10].

However, these approaches are primarily aimed at hiding application data from third parties. In our work, we suggest an approach that allows other processes to read memory of a trusted process, but denies to modify it. Such assumption is useful for cases when trusted application needs to pass some data to other processes by means, for example, of shared memory. That is, our approach protects execution process of a trusted application, but does not hide its whole data from other programs. But if necessary, our system can be easily modified to completely deny access to application's memory.

An advantage of Overshadow is that no modifications are required in existing software (OS, applications) and hardware. More precisely, there are no hardware-specific requirements only if hypervisor used is able to perform virtualization without hardware assistance. However, in this case protection system architecture is bounded to architecture of particular hypervisor. Moreover, such hypervisors for x86 platform are rather complex and they are rather hard to implement (since x86 architecture by itself is hard to virtualize due to design). Among effective implementations, we can mention only VMware VMM (used in Overshadow) and VirtualBox [11]. Since such hypervisors are complex (and in addition, VMware hypervisors are mostly closed source products), it is not easy to modify them to implement additional functionality.

On the other hand, in the last several years Intel and AMD have added virtualization support to their processors and made it easier to create virtualization products [12]. These possibilities are now utilized by such products as Kernel-based Virtual Machine (KVM), Xen, VMware ESX and others. In our approach we assume that target system provide hardware-assisted virtualization. This puts some limitations on hardware where our approach is applicable, but significantly simplifies its implementation.

III. Controlling Consistency of a Trusted Process

In our threat model, we suppose that the operating system is not reliable and contains vulnerabilities which can be exploited by malware to gain high privileges. Such privileges would allow attacker to control all processes running in the system and perform malware injections in their files or directly in the process code in the memory.

In our protection system, potentially compromised OS is located inside virtual machine controlled by hypervisor, which is a core part of the protection system. Hypervisor has higher privileges than OS inside VM and can monitor and control events inside VM.

In order to guarantee consistency of a trusted application, hypervisor should guarantee the following:

- application files on disk (in particular, executables and libraries) are not modified by malicious software; in this paper, we only consider executable files and libraries that form the application, ignoring the task of protection of other files and resources that can be used by application (e.g., protection of different data files);
- address space of a running process is not modified in an unallowed way by other processes running in OS.

Let us consider how these tasks are solved by suggested protection system.

A. Checking Consistency of Executable Files and Libraries

When launching a trusted application, we should first ensure that executable being launched is an expected one. In order to do this, we should check that application executable file (and shared libraries, if any) on disk was not modified by malicious code. To make such check possible, every trusted application in our system should provide hypervisor with a *registration data*, generated inside trusted environment on the basis of application files. This registration data is stored in hypervisor and cannot be accessed by OS.

Registration data for application executable files and shared libraries consists of SHA-1 hash codes. Such codes are generated for every memory page containing either instructions or static data. Currently we assume that the page size is equal to 4 kilobytes (a default value on most systems). However, nowadays Linux provides support for larger pages [13], and we plan to support such pages in future, as well.

B. Protecting Control Flow

In our system, the working virtual machine is provided with a single-core virtual CPU, so OS inside this VM can only use a pure time-sharing multitasking. There is no way to run different processes on different CPU cores in parallel. In particular, at any moment of time either CPU and other resources are used by trusted code or they are used by potentially malicious software. Thus, if we want to protect trusted process, we should only ensure that the process address space and other system resources that can influence process execution (e.g., different system registers) were not modified in a forbidden way while the trusted process was inactive. When trusted process is active, all events in the system are allowed. In particular, trusted process can modify its own code segments in memory. Besides application code and static data loaded at launch, we can control consistency of any data pages loaded by during process execution. More particular, we track states of all memory pages written by the process.

In order to implement such protection, we use hypervisor to handle interruptions of trusted code execution. When a trusted

process is interrupted, hypervisor saves information about its address space and other protected resources inside its own memory. Only after that, control is passed to operating system. When OS returns control to the trusted process, hypervisor compares actual state of protected resources with the saved one. If any discrepancy is detected, the protection system reports an attack attempt and the process is not considered to be trusted any more. From that moment, it will not be allowed to use protected system resources (e.g., network connection).

One of the main components of the protection system is a *register integrity checker* used to protect control flow of trusted processes. The control flow is considered to be integral, if the following requirements are met:

1) actual address of program entry point is equal to the value specified in the registration data;
2) every time the control is passed from the OS kernel to the trusted process, address of instruction invoked in the process is either equal to the instruction where the process was previously interrupted, or is equal to a special signal handler (registered by the process in advance).

The first requirement is checked only when the process is launched using a system call like $exec()$. More generally, it should be checked when the process enters the trusted mode, but in our work we do not consider situations when the process can enter the trusted mode after the launch. The second requirement is checked every time the control is passed to the trusted process. When such an event occurs, the hypervisor verifies instruction address, as well as values of general purpose, segment and different system registers.

C. Protecting Address Space

Hypervisor controls integrity of all virtual memory pages (containing either code or data) of the process. When a trusted process accesses a memory page for the first time, this page is marked as *active*. If the page accessed for the first time contains program code or static data, then it is checked that the page hash sum corresponds to the one specified in the registration data. This allows to verify that the program code and static data were not modified after registration data was generated. Other pages are allowed to have random data when they are accessed by trusted process for the first time. If in the sequel trusted process accesses such a page, the hypervisor checks that the page content was not modified since the last time when it was accessed by the process.

In order to perform such integrity monitoring, hypervisor uses a special control table of process active virtual pages which we call *Memory Integrity Table* (MIT). For every virtual page V, the MIT table contains either address of corresponding physical page P (if V is mapped to a physical memory) or hash sum H if the page is not mapped.

At runtime, programs can detach memory pages from their address space (e.g., by means of $munmap()$ system call). Hypervisor tracks such system calls and removes the *active* mark from the detached pages.

Pages storing dynamic data inside address space of a trusted process can be subjected to legal modifications by the process itself, as well as by some system calls (e.g., $read()$). If an active page of a trusted process is mapped to a physical page, then write access to that page is allowed for the trusted process only. When a trusted process tries to access a page for which a hash sum is set in the MIT table, hypervisor checks integrity of that page by calculating hash code for its current content and comparing it with the expected value stored in the control table.

Moreover, hypervisor allows only modifications that touch memory areas explicitly specified in the system call parameters. Modifications outside such explicitly specified areas are prohibited. It is important to note that on Intel x86 architecture it is possible to set access permissions on the page-level basis, while processes may want to write data which is not aligned to page size. In order to support protection of such data, hypervisor used special trick based on the fact that for every process one can specified address area writable for kernel with per-byte precision.

Before transferring system call to OS kernel, the protection system for every out parameter allocates a ”shadow” memory area inside virtual address space of the process and set registers controlling passing of return values to point to that area. Thus, output of every system call is redirected to memory area not used by the process. When system call returns control to the process, hypervisor copies its output to corresponding areas inside process memory.

In order to maintain mappings in the MIT table and to intercept page access attempts, hypervisor runs every trusted process in a separate *protection domain*. Protection domain is a set of memory pages with individual access permissions. This set of pages for a particular protection domain is dynamically altered by hypervisor when process requests more memory or frees unnecessary pages. Every attempt to access a page outside the protection domain, as well as access violation for the page inside the domain, leads to exception which is caught and handled by hypervisor.

Implementation of protection domains is based on the *Nested Page Tables* (NPT) mechanism (NPT implementation in Intel processors is called *Extended Page Tables*, the one from AMD – *Rapid Virtualization Indexing*). NPT tables are used to perform translation of pseudo-physical addresses used inside VM to physical addresses of the real hardware. When a process is launched in the trusted mode, hypervisor creates an empty set of NPT tables for it. Every time when OS kernel passes control to the trusted process, hypervisor activates page tables corresponding to that process. This is performed by means of the *Virtual Machine Control Block* (VMCB) structure. When trusted process is interrupted and control is passed back to the OS kernel, hypervisor switches active nested pages once again and activates tables of untrusted domain (a joint domain for OS kernel and other untrusted processes).

When a process tries to access a page which is not yet reflected in the NPT tables, or when access violation occurs, a

Nested Page Fault (#NPF) exception is thrown, VM is stopped and control is passed to the hypervisor. Hypervisor maintains NPT mapping only for active pages which are not swapped out to the storage device and which were not modified by third-party processes. This approach allows to determine if the process accesses a page for the first time or it accesses pages which were modified since the last access by this process or loaded from swap.

When the #NPF exception is thrown, a pseudo-physical address of page inside VM is reported, access to which led to the exception. However, in order to get the expected hash sum for the page from the control tables, hypervisor should also know a virtual address, access to which finally led to #NPF. In order to calculate virtual address, hypervisor disassembles the current instruction of the trusted process (address of such instruction is always stored in the IP register) and analyzes all virtual addresses accessed by this instruction. Using page table of the operating system, hypervisor calculates real addresses corresponding to these virtual ones and detects which of them corresponds to the pseudo-physical address access to which led to the #NPF exception. With this virtual address, the hypervisor is able to verify integrity of the page accessed by the trusted process.

D. Protecting Dynamically Linked Applications

Address space protection approach described above easily applies for statically linked programs. Such a program is represented by a single executable file that does not import any libraries from the OS, so we can know in advance location of code and static data inside the application. However, nowadays developers often take an advantage of using splitting functionality between separate libraries which are combined together by the loader during program start up or even loaded by request during program execution (such functionality is provided in Linux by **libdl** library). Protection of such programs (especially those that use libdl functionality) introduces new challenges.

Dynamically linked application consists of a main executable file and several libraries loaded by dynamic loader during application launch. In Linux, for both executable files and dynamic libraries ELF format is used. Every ELF file has a set of DT_NEEDED entries which store names of libraries that should be loaded with this file. When launching an executable, dynamic loader processes DT_NEEDED entries of the file itself, then DT_NEEDED entries of libraries loaded as file dependencies and so on – such iterations are performed until DT_NEEDED entries of all files from the loaded set are satisfied by files from this set.

The set of DT_NEEDED entries can be extracted from the ELF file by means of appropriate tools. However, the final set of loaded libraries can be different for the same executable in different Linux distributions, because internal dependencies of libraries can differ. Moreover, in addition to dependencies statically recorded in the ELF file structures which are resolved during file launch, it is possible to load libraries at runtime by means of functions provided by libdl library. In many cases, it is almost impossible to automatically detect which libraries will be loaded using such functionality, because the name of the library to be loaded can be calculated at runtime.

Due to these facts, in our approach user should explicitly list all the libraries that will be loaded during application work in particular system. This set considered to be a set of trusted files. If a library not included in this set will be loaded and put to the application address space, this will be reported as an attack attempt.

In addition to libraries, for every dynamically linked application the Linux kernel exposes a shared object called *Virtual Dynamically-linked Shared Object* (VDSO) which exports symbols implementing virtual system calls [14]. Traditionally, system calls in Linux on the x86 platform were implemented using 0x80 software interrupt. With modern processors, faster implementations are available that use SYSCALL or SYSENTER instructions for AMD and Intel processors respectively. For every of these techniques, the Linux kernel has a corresponding VDSO variant. Implementation of all these three VDSO variants can be extracted from the Linux kernel sources.

Thus, a memory image of a dynamically linked application consists of the following components:

- executable file (launched by user or by other process);
- dynamic loader (usually – ld-linux.so);
- set of libraries specified as ELF file dependencies and loaded at application start;
- set of libraries loaded at runtime using libdl functionality;
- VDSO library.

Registration data of dynamically linked application should contain information about all these components.

An important feature of dynamic libraries is that their code is position-independent and can be located any area of application's address space. Address value specified in the ELF file header in Linux running on x86 platform nowadays is just a recommendation for the loader. In reality, dynamic loader can place every file at other address, and such addresses can vary in different systems or even in different instances of the same application.

Note that since VDSO is a shared object, it can also be located at any address inside process address space. Thus, location of VDSO in process memory can be different for different processes.

Finally, executable files can also contain position-independent code. Executable files that consist of such code only (*Position-Independent Executables*, PIE) are relatively widespread in the Linux ecosystem.

Thus, every component of dynamically linked application can be located at any virtual address inside application address space. Since the memory is allocated and managed on the per-page basis, correlation between actual address of every component and the value specified in ELF header is expressed by the following formula:

$$Actual\_address = ELF\_address + k * (page\_size)$$

where k is some integer number.

Thus, though location of different components of dynamically linked application in virtual memory is not known in advance, these locations can be easily calculated by hypervisor during application start up. Location of libraries loaded using libdl functions can be calculated at the moment when $dlopen()$ function is invoked. This allows to adopt registration data for every particular launch of application. As we will discuss later, the only thing hypervisor has to calculate is a difference between real address and the value specified in the ELF header (and thus reflected in the registration data) which is identified by a single integer number k. It is important to note that since library code is position independent, it is not subjected to any modifications by loader.

Before passing control to entry point of dynamically linked application, dynamic loader should link together all components of application and set actual addresses of all imported symbols. For the context protection system, it is important to ensure that no malicious code interfere with this process, replacing address of legal imported function with address of malicious symbol. Let us proceed with details of dynamic loader work process and see how protection system guarantees consistency of function addresses.

During dynamic linking process, system loader first loads all necessary files to memory and then initiates symbol resolution process. For every binary symbol imported by some ELF file (this file is called *importer*) the loader should locate the file where the symbol is implemented (this file is called *exporter*). Dynamic loader analyzes symbol tables of exporter and importer and updates the Global Offset Table (GOT), which is located at the data segment of the importer. The GOT table contains an entry for every imported binary symbol (corresponding to a function o global variable). Symbol resolution procedure is the same for libraries loaded during application launch and the one loaded at runtime using libdl functionality.

Code segment of ELF file that imports some functions contains Procedure Linkage Table (PLT) which contains a *stub* symbol for every imported function. When an attempt is performed to call some imported function, the control is passed to the corresponding stub which takes unnecessary address from the GOT table and passes control to that address. Thus, a call to an imported function is an indirect call by address recorded in the appropriate GOT table entry.

GOT and PLT tables used when a call to imported function happens are located in the segments of that ELF file from which the call is performed. Thus, they are taken into account when file hash sum is calculated and monitored by the protection system during application work. The whole dynamic linking process, including modification of the GOT table, is performed by the dynamic loader which works in the user space. The process requires no kernel-level activities and thus cannot influence other applications in case of errors. This is one of the advantages of using ELF format for executable files.

Thus, if dynamic loader is a trusted program, then all the actions during dynamic linking are performed by trusted code. Dynamic loader is much more smaller then the Linux kernel and it does not vary significantly among different distributions (in particular, it does not allow insertion of some third-party software such as drivers in its code). Thus, we believe that it is reasonable to consider dynamic loader to be a trusted process. In the rest of the paper, we use the assumption that the dynamic loader is a trusted program. Note that since dynamic loader is included in the process image, hypervisor is able to compare its content with registration data. Thus, the protection system is able to check that the loader is the same as the one in the system where the registration data was generated.

When launching a dynamically linked application, the Linux kernel creates a virtual address space for the new process and loads application executable file (which was actually launched), dynamic loader and VDSO library there. All other libraries are loaded using explicit calls to the $mmap()$ system call from the dynamic loader. This call returns a virtual address where the library is located. Since hypervisor monitors all system calls performed by application, it can track library loading and build a mapping between library name (passed as a parameter to the $open()$ call, whose result is then passed to $mmap()$) and library location in the process address space. For every loaded library it is checked, if the library is included in the list which was provided by user when generating registration data. If so, then hypervisor is able to compare hash sum of the loaded library with the expected value and verify that this is, indeed, an expected file. If the loaded file is not included in the list of trusted libraries, or if its hash sum does not match the value expected, then the loaded code reported to be untrusted. If the control is passed to such code, then hypervisor will nullify application privileges, so it will not be considered to be trusted any more.

OS kernel passes addresses of components loaded during application launch to dynamic loader using *ELF auxiliary vectors* [15]. During application launch, array of such vectors is put at the process stack just after environment variables and thus can be easily analyzed by hypervisor. Each vector is just a pair of numbers (vector *type* and *value*).

In order to be able to protect application address space, we should know values of vectors with the following types:

- AT_PHDR – Base address of executable file;
- AT_ENTRY – Entry point of program ;
- AT_BASE – Base address of dynamic loader;
- AT_SYSINFO_EHDR – Base address of the VDSO library.

When application is launching, the control is first passed to the entry point of the dynamic loader. The loader performs dynamic linking of the executable file launched and libraries loaded as its dependencies and then passes control to the address specified in the AT_ENTRY vector. AT_SYSINFO_PHDR vector is used by libc library to perform a system call.

Every component of dynamically linked application can be loaded at (almost) random address which differs from the one recorded in the file header. However, in any case the following conditions are met:

- For every loaded component, hypervisor can obtain the effective address where the component is located in

virtual memory before the control is passed to that component or before that component is accessed by someone else (e.g., by dynamic loader which should at least read header of loaded file during dynamic linking process). Thus, hypervisor can verify component integrity before the component is used by other parts of trusted application.
- Every file is stored in the address space continuously, so if a file is loaded at the address different from the one specified in its header, hypervisor just have to shift registration data for this file, without a need to recalculate it.

Thus, hypervisor is still able to protect application address space, but for every application component it should calculate an effective address where the component is loaded. For executable files, dynamic interpreter and VDSO library such addresses can be obtained from AT_PHDR, AT_BASE and AT_SYSINFO_EHDR vectors respectively. For other libraries, the effective address is a result of the $mmap()$ system call.

After calculation of real address value, hypervisor adjusts registration data for the trusted process by updating the MIT table which stores mapping between addresses of every page of trusted process and hash codes. For every component, such an update is performed as soon as component virtual address becomes known. For shared libraries, this happens after return from the $mmap()$ system call (for libraries specified directly as ELF file dependencies – during application start up, for libraries loaded using $dlopen()$ function – when loading a library during application work). For other components the update happens at the moment of application start up (more particular – after return from the $execv()$ system call).

Information about real location of application components is obtained from the Linux kernel, which is untrusted in our threat model. However, if kernel provides hypervisor with wrong information (that is, real address values differ from the one reported by kernel), this will be detected as soon as some part of application will try to access a page with wrong data. In this case, expected hash sum for the page will differ from the observed one and the attack will be reported. Thus, it is impossible for the kernel to substitute some part of the application without being noticed.

It is important to note that similar to hypervisor, dynamic loader during the linking process uses information obtained from the kernel. Protection system should guarantee that the loader uses the same data as the hypervisor itself – otherwise the resulting application image in memory can differ from hypervisor expectations. Thus, we should guarantee that hypervisor and dynamic loader use the same values of auxiliary vectors. But these vectors are located in the application stack which is monitored by our protection system and whose integrity is guaranteed. Thus, hypervisor is able to control that the loader itself (ld-linux.so) and application components (executable and libraries) are consistent and match registration data.

Thus, protection of stack, code segment and process data automatically guarantees protection of PLT and GOT tables, as well as ELF auxiliary vectors.

In order to prevent execution of unauthorized code, hypervisor uses one more feature of modern hardware architectures, namely NX (No eXecute) bit. If a process tries to execute instruction from a page marked with this bit, a page fault exception is thrown. Hypervisor sets NX bit in the NPT for all pages of a trusted process except those that contain executable code authorized by means of registration data. If a trusted process attempts to execute a code from a NX page, a page fault exception is thrown. In case of page fault, hypervisor checks error code and if the fault was caused by attempt of launching some instruction, then the process is considered to be compromised and protection system deprives this process of privileged rights, so it is not trusted any more.

E. Protecting MultiThread Applications

In addition to dynamically linked programs, nowadays many applications use multithread paradigm, when program consists of several threads which work in parallel in the same address space. In Linux, threads are created using $clone()$ system call with $CLONE\_VM$ flag.

All threads of the same process have the same page tables. For such threads, hypervisor also uses the same nested page tables and the control MIT table. It is important that in our system a virtual machine where the trusted processes work has only one CPU core available, so at any time point only one thread can really execute CPU instructions. Thus, when considering address space access, we can safely ignore the fact that the process is divided on several threads. In a single-core system, it does not matter if the access is performed from mono-thread trusted process or from some thread of multithread trusted process. In addition, absence of "real" parallelism allows us not to care about synchronization of thread access to nested page tables or MITs.

However, every process thread in Linux can have its own address space to store data unique to particular thread. This storage (called *thread local storage*) is created automatically by compiler for variables that have $\_\_thread$ specifier, or can be created at runtime by means of functions like $set\_thread\_area()$.

Thread local storage is implemented completely on the software layer by compiler, libc library and Linux kernel. For every thread, a separate memory area is allocated to store thread-specific data. Since all this memory areas are allocated inside process address space (common for all threads), every thread can potentially access a local storage of any other thread.

Thread local storage is located in a separate segment managed by the GS register which is set to different values for different threads when thread-local data is accessed. In order to control integrity of thread local data, the memory protection system should monitor GS register value and corresponding record in the segment table. These values can be monitored in the same way as other resources and processor registers, so protection of multithread applications fits well the approach used in our protection system.

IV. Implementation

The approach suggested in this paper was implemented on the basis of KVM (Kernel-based Virtual Machine) hypervisor which is included in the Linux kernel. KVM itself is a kernel module which adds hypervisor functionality to the Linux kernel. KVM requires QEMU application to manage and emulate different virtual machine resources and devices (keyboard, network card, etc.). In our work, we use KVM version 88, kernel 2.6.31.6 and QEMU 0.13.0. Virtual machines with trusted applications are run under Fedora Linux 13 with the same kernel (2.6.31.6). Currently our implementation supports virtualization of 32-bit systems on AMD platform. For the experiments described below, we have used AMD Phenom 9750 Quad-Core Processor which has four 2.4MHz cores. The host machine had 4GB RAM, and virtual machine was configured with 512MB RAM. It is important to note that it is not necessary to load hypervisor when the machine starts; we have investigated possibility of on-demand activation of protection system [16]. Such on-demand activation (which involves launching hypervisor from running OS, creation of a virtual machine and placing the OS inside this machine) is possible, though requires special hardware (Trusted Platform Module, TPM).

To automate generation of registration data for trusted processes, we provide a tool named ElfHash that processes executable file which will be launched and its DT_NEEDED dependencies. We suppose that the system where the registration data is generated provides trusted versions of libraries used by dynamically linked application. On the basis of such assumption, the ElfHash tool analyzes system libraries that satisfy application dependencies and create registration data for them. Alternatively, user can provide the tool with his own versions of such libraries. In additional, user can specify libraries not mentioned in file dependencies but that can be loaded at runtime using libdl functionality. Finally, we provide registration data for the dynamic loader itself. The data is generated using VDSO implementation which is considered to be trusted.

A. Attack Detection

In order to evaluate if the protection system works as expected, we have emulated two kinds of attacks on trusted processes: modification of application files on disk and modification of trusted code in memory. For our experiments, we have used the SSH tool which was establishing connection from the virtual machine to some remote host. Network card was considered to be a protected system resource, so only trusted applications were allowed to access it. Operating system itself was not aware of network card.

When emulating attacks concerning application file modifications, we have investigated behavior of the protection system in the following cases:

- SSH executable differs from the one registered in the protection system;
- one of the libraries from ELF DT_NEEDED dependencies differs from the one used to register SSH in the protection system;
- one of the libraries loaded by SSH using $dlopen()$ differs from the one registered in the protection system.

In the first two cases, the protection system reported the attack attempt during application startup. In the third situation, attack attempt was reported at the moment of $dlopen()$ call. In all cases, access to network card was denied and connection to remote host was not established.

In addition, we have checked situations when library is loaded which is not present in registration data. The library can either be loaded by the process itself or pre-loaded if user sets LD_PRELOAD variable. In such situations, the protection system also reported attack attempt, as expected.

To emulate attacks concerning modification of the code of a running process, we have used techniques based on the $ptrace()$ system call. In particular, we have used the PreZ tool [17] which attaches to running process and creates its own thread inside it. This thread opens a port for TCP connections and spawns a shell for every incoming connection. The shell can then be used by remote party to perform different actions on the machine with privileges of the infected process.

During our experiments, the protection system has successfully detected all code injection attempts and blocked access to the network card for the SSH process.

B. Performance

In order to estimate delays introduced by the protection system, we have compared performance of two applications – Apache web server and SSH client – in the following cases:

- applications are launched on bare hardware;
- applications are launched inside virtual machine without protection systems;
- applications are launched inside virtual machine with protection system controlling their address space.

In order to measure Apache performance, we have used the Flood load tester (a part of the Apache project). Number of processes launched by Apache to serve the requests (that is, number of trusted processes) was limited to ten. In case of SSH, we have used the SCP utility to copy large (four gigabytes) file through network. There was only one trusted process in this experiment.

Since our protection system assumes that only one processor core is assigned to virtual machine, in our experiments we were using a single core in all cases. Measurement results are presented at Fig.1. We have normalized the results and assigned 100 units to performance on the bare hardware, so it is easy to compare the measurements.

As one can see, in case of Apache performance loss is almost unnoticeable, while for SSH it is much more higher. This is probably caused by the fact that during the experiment Apache was receiving simple requests and their processing did not require much memory, while copying file with SSH involved encryption of large amount of data which led to significant usage of memory by trusted process. In addition,

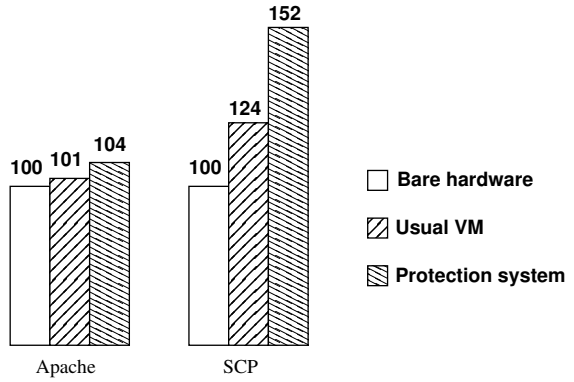

Fig. 1. Protection system performance

when dealing with large data transferring through network, a significant delay is introduced by emulation software by itself. Nevertheless, even for SSH performance loss is acceptable if sender wants to protect the data from potentially compromised OS.

V. Conclusion

In this paper, we have presented a novel approach for protecting applications running inside potentially compromised operating system. The approach is based on using virtual machine monitor (hypervisor) which tracks all events inside OS and prevents unallowed modifications of application resources.

Unlike other hypervisor-based protection techniques, our approach does not lead to isolation of application from other OS components. Hypervisor's functionality is flexible and can be adjusted to control usage of any particular hardware resources, so only trusted applications have access to them. For example, application of our protection system to control usage of network connection is described in [18]. Finally, the approach can be extended in order to protect all application files, not only executables and libraries. This will require interception of direct file input/output (using $read()/write()$ system calls) in addition to $mmap()$ manipulations and storing hash codes for all files used by application. We believe that it is not hard to extend our approach in this way, though such improvements can introduce significant performance drop.

The approach does not require any modifications in operating system or applications, but relies on several aspects of hardware-assisted virtualization implemented in Intel and AMD x86 processors. In order to implement the approach, there is no need to develop a hypervisor from scratch. Instead, one can extend existing products such as KVM or Xen. Our KVM-based implementation has demonstrated that performance overhead introduced by the protection system is acceptable, so we believe that the approach is viable and can be applied in those areas where information security is the primary goal.

References

[1] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," *ACM Trans. Comput. Syst.*, vol. 23, no. 1, pp. 77–110, 2005.

[2] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler, "An empirical study of operating system errors," in *SOSP*, 2001, pp. 73–88.

[3] G. Kroah-Hartman, "How linux supports more devices than any other os, ever," *O'Reilly Media Interview*, Oct. 2008. [Online]. Available: http://broadcast.oreilly.com/2008/10/how-linux-supports-more-device.html

[4] A. S. Tanenbaum, J. N. Herder, and H. Bos, "Can we make operating systems reliable and secure?" *Computer*, vol. 39, pp. 44–51, May 2006. [Online]. Available: http://portal.acm.org/citation.cfm?id=1137232.1137291

[5] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: a virtual machine-based platform for trusted computing," *SIGOPS Oper. Syst. Rev.*, vol. 37, pp. 193–206, October 2003. [Online]. Available: http://doi.acm.org/10.1145/1165389.945464

[6] R. Ta-Min, L. Litty, and D. Lie, "Splitting interfaces: making trust between applications and operating systems configurable," in *Proceedings of the 7th symposium on Operating systems design and implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 279–292. [Online]. Available: http://portal.acm.org/citation.cfm?id=1298455.1298482

[7] J. S. Dwoskin and R. B. Lee, "Hardware-rooted trust for secure key management and transient trust," in *Proceedings of the 14th ACM conference on Computer and communications security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 389–400. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315294

[8] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang, "Architecture for protecting critical secrets in microprocessors," *SIGARCH Comput. Archit. News*, vol. 33, pp. 2–13, May 2005. [Online]. Available: http://doi.acm.org/10.1145/1080695.1069971

[9] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems," *SIGOPS Oper. Syst. Rev.*, vol. 42, pp. 2–13, March 2008. [Online]. Available: http://doi.acm.org/10.1145/1353535.1346284

[10] J. Yang and K. G. Shin, "Using hypervisor to provide data secrecy for user applications on a per-page basis," in *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ser. VEE '08. New York, NY, USA: ACM, 2008, pp. 71–80. [Online]. Available: http://doi.acm.org/10.1145/1346256.1346267

[11] Oracle vm virtualbox. [Online]. Available: http://www.oracle.com/us/technologies/virtualization/oraclevm/061976.html

[12] J. Fisher-Ogden. (2006) Hardware support for efficient virtualization. [Online]. Available: http://www.cse.ucsd.edu/~jfisherogden/hardwareVirt.pdf

[13] R. Krishnakumar, "Hugetlb - large page support in the linux kernel." *Linux Gazette*, vol. 155, Feb. 2008. [Online]. Available: http://linuxgazette.net/155/krishnakumar.html

[14] J. Petersson, "What is linux-gate.so.1?" Aug. 2005. [Online]. Available: http://www.trilithium.com/johan/2005/08/linux-gate/

[15] M. Garg, "About elf auxiliary vectors." 2006. [Online]. Available: http://articles.manugarg.com/aboutelfauxiliaryvectors.html

[16] D. Yefremov and P. Iakovenko, "An approach to on-demand activation and deactivation of virtualization-based security systems," in *Proceedings of the fourth Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE 2010)*, 2010, pp. 157–161. [Online]. Available: http://syrcose.ispras.ru/2010/files/syrcose10_submission_5.pdf

[17] F. Loukos. (2010) Injecting code at a running process. [Online]. Available: http://fotis.loukos.me/blog/?p=145

[18] I. Burdonov, A. Kosachev, and P. Iakovenko, "Virtualization-based separation of privilege: working with sensitive data in untrusted environment," in *Proceedings of the 1st EuroSys Workshop on Virtualization Technology for Dependable Systems*, ser. VDTS '09. New York, NY, USA: ACM, 2009, pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/1518684.1518685