VCCBox: Practical Confinement of Untrusted Software in Virtual Cloud Computing*

Jun Jiang, Meining Nie, Purui Su, and Dengguo Feng

Trusted Computing and Information Assurance Laboratory,
Institute of Software, Chinese Academy of Sciences,
Beijing 100190, China
{jiangjun,niemeining,supurui,feng}@tca.iscas.ac.cn

Abstract. Recent maturity of virtualization has enabled its wide adoption in cloud environment. However, legacy security issues still exist in the cloud and are further enlarged. For instance, the execution of untrusted software may cause more harm to system security. Though conventional *sandboxes* can be used to constrain the destructive program behaviors, they suffer from various deficiencies. In this paper, we propose *VCCBox*, a practical sandbox that confines untrusted applications in cloud environment. Leveraging the state-of-the-art hardware assisted virtualization technology and novel design, it is able to work effectively and efficiently. VCCBox implements its system call interception and access control policy enforcement inside the hypervisor and create an interface to dynamically load policies. The in-VMM design renders our system hard to bypass and easy to deploy in cloud environment, and dynamic policy loading provides high efficiency. We have implemented a proof-of-concept system based on Xen and the evaluation exhibits that our system achieves the design goal of effectiveness and efficiency.

Keywords: Sandbox, Hypervisor based security, Hardware assisted virtualization, Cloud computing.

1 Introduction

In recent years, cloud computing has become a heated topic in both industry and academia. Virtualization, as an underlying technology of cloud computing, plays a key role in utility computing and private cloud. Among all virtualization techniques, hardware assisted virtualization has been widely adopted since it is compatible with existing OS kernels and is supported by various commodity and open-source hypervisors.

Cloud computing is a double-edged sword from the perspective of security. It provides better environment for solving security problems but also enlarges the

* This work is supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 61073179, National Basic Research Program of China (973 Program) under Grant No. 2012CB315804, and Natural Science Foundation of Beijing under Grant No. 4122086.

harm of legacy security issues. For example, some programs may behave maliciously while providing desired features, and this could be either intentional or not. How to securely execute untrusted applications and confine their destructive behaviors has been an everlasting issue in system security. Generally, this problem can be resolved by *sandbox*, a mechanism that controls the runtime environment of a program and mediates its interactions with the outside. Hence, the program behavior can be limited to what the user allows. Unfortunately, most currently available sandboxes possess various deficiencies, such as liability to be bypassed, and requirement of modified or dedicated kernel. More importantly, as we step into the cloud computing era, it becomes difficult and even impossible to deploy them in real production environment. We present a detailed examination of representative sandbox mechanisms in Section 6.2.

To overcome such shortcomings, we present *VCCBox*, a sandbox architecture constructed on top of the hypervisor, which embraces contemporary hardware assisted virtualization technology for robustness and ease-of-deployment. We observe that the system call is the only entry for an application to perform sensitive operations and access system resources. Hence, we intercept system calls from the hypervisor level and check whether they violate access control policies that are compiled from policy scripts written by the user in a C-like language and loaded into the hypervisor dynamically at runtime. The decision made to a system call can be either *permitted*, *disallowed* or *deceived.*

In summary, we make the following contributions:

– We *first* propose a sandbox architecture based on hardware assisted virtualization technology, which overcomes several defects of existing solutions.
– We have implemented a mechanism to dynamically load code into the hypervisor at runtime. To the best of our knowledge, we are the *first* to use this technique in hypervisor-based security mechanisms.
– We have devised a special variant of C programming language as the policy description language, which enables fast development of effective, flexible and powerful policies.
– We have implemented a Xen-based prototype system named VCCBox and performed detailed evaluation showing that our system is effective and efficient for adoption in production cloud environment.

The remainder of this paper is organized as follows. Section 2 presents not only the application scenario and technical background of our system, but also the design of the system, while Section 3 details the implementation. Then we present the results of evaluation in Section 4 and analyze possible limitations of our system and some future work in Section 5. Finally, we discuss several related work in Section 6 and conclude our paper in Section 7.

2 System Overview and Design

In this work, we utilize the contemporary hardware assisted virtualization technology to design a sandbox mechanism named VCCBox. We take advantage of the higher privilege of the hypervisor to achieve non-circumventable protection. Moreover, hardware assisted virtualization is prevalent in cloud computing nowadays, rendering our system suitable for real production environment.

2.1 Application Scenario

To demonstrate the usefulness of our VCCBox system, we provide the following two scenarios:

- Peter is an administrator of several virtual servers running a few services such as HTTP and FTP in a company. However, he is not sure whether these server programs contain malicious parts. It is possible for a web server to possess a backdoor that sends out sensitive files when triggered by a special URL.
- James is a manager of a few virtual private servers, and he rents his virtual machines to individuals. Unfortunately, some tenants use the VMs to perform malicious activities such as sending spams or launching DDoS attacks. It can be trivial and time-consuming to manually stop these behaviors.

In above scenarios, sandboxes can be used to confine the vicious behaviors of untrusted or malicious programs. However, traditional in-OS sandboxes may not fulfill this need. In the first scenario, it could be a tedious and non-trivial task to install and configure sandboxes for respective virtual servers considering the large amount of VMs on a physical machine. Furthermore, in-OS sandboxes are not applicable to the second scenario at all, since the end user has full control on the virtual machine and can easily disable the sandbox. Under such circumstances, our VCCBox system is useful, since it runs at the hypervisor level, and thus is easy to be deployed in cloud environment and hard to bypass.

2.2 Technical Background

Since our approach involves hardware assisted virtualization, specifically, Intel VT, we give a brief introduction of it. The traditional x86 architecture has four privilege levels, ring 0 (highest) for kernels to ring 3 (lowest) for applications. In Intel VT, these four rings are categorized into *VMX non-root mode*, and a new *VMX root mode* with even higher privilege is introduced. Since the legacy four rings still exist, the operating system can run without modification. When virtualization is enabled, several sensitive instructions change their semantics and trap into the hypervisor for the whole system to run correctly. Namely, the virtual machine and the hypervisor alternately obtain the CPU time slice. When the operating system tries to execute a sensitive instruction or an external interrupt occurs, an event called *VMexit* is triggered. Therefore, the processor switches its privilege level to VMX root mode and a previously registered *VMexit handler* is called to enable the hypervisor to deal with the event appropriately for correct virtualization. After that, *VMentry* gives the control back to the operating system and reverts to the previous status. Therefore, the hypervisor naturally provides security applications with an opportunity due to its higher privilege and its capability to intervene in the execution of the virtual machine systems. Currently, modern processors equipped with this technology are widely used in production environment.

Assumption. Since our method involves virtualization, we assume that the hypervisor and the management domain are trusted. We believe that this assumption is reasonable since it is a fundamental assumption shared by many other hypervisor-based security mechanisms [8,14,16,24] and can be consolidated by existing hypervisor protection techniques [1,21,23]. Though attacks to the VMM exist, they are out of the scope of our work.

2.3 System Architecture

VCCBox is constructed on an existing hypervisor in order to be compatible with the production environment. As is illustrated in Fig. 1, our system resides in both the hypervisor and the management domain and is composed of four parts: policy manager, policy library, system call interception and system call feedback. The asterisks indicate that the corresponding parts can be multiple.

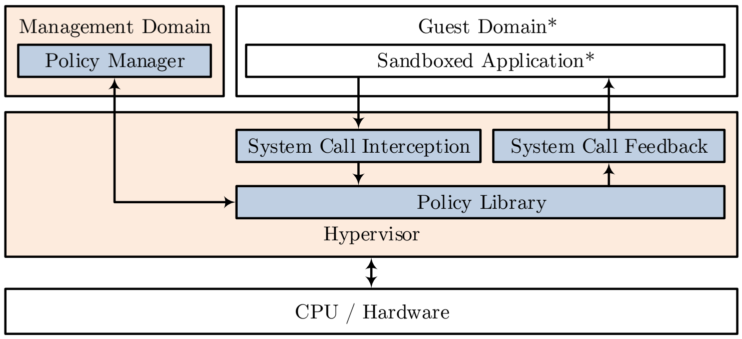


Fig. 1. Architecture of VCCBox

VCCBox works in a straightforward way. It intercepts system calls since they are essential for an application to access system resources, and then consults the policy library to find out applicable measures. Leveraging the higher privilege of the hypervisor and proper design, our system call interception mechanism cannot be bypassed. Moreover, the policies in the library can be added, removed and updated dynamically at runtime, eliminating domain-hypervisor context switches and making our system highly efficient. The policies are compiled from flexible policy scripts written by the user of our system (usually an administrator) using a C-like policy description language. Fig. 2 depicts the entire system workflow.

System Call Interception and Feedback. System call interception is the first step towards sandboxing. Currently, two approaches can be used by an application to request system services, i.e., software interrupt and fast system call mechanism. However, the hypervisor cannot intercept them directly since neither of them triggers *VMexit* events. Fortunately, there are multiple approaches which allow the system calls to be intercepted *indirectly*. Once a system call is intercepted, we look for the corresponding policy and enforce it. The policy uses virtual machine introspection [8] to obtain necessary information and makes a decision out of the following three: *permitted*, *disallowed* and *deceived*. To feed back the result, we take the following approach: nothing additional is required for *permitted*; for *disallowed* and *deceived*, we "skip" the system call and provide appropriate return value to the application by modifying relevant registers; one extra thing is needed for *deceived*, i.e., filling the "output" parameters of the system call with specific value.

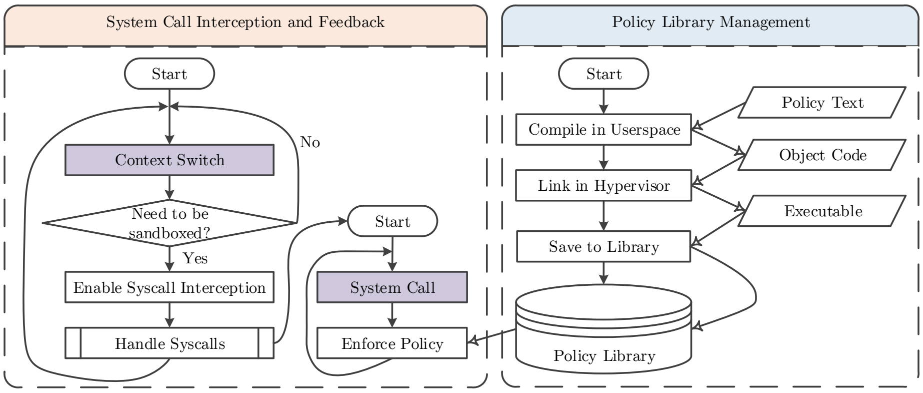


Fig. 2. Workflow of VCCBox

Policy Library Management. Our policies are per program. Thus, we check if the upcoming process needs to be sandboxed when a context switch occurs. If not, system call interception will not be enabled in order to retain the performance. Otherwise, each interesting system call of the sandboxed process is associated with a policy routine. The policies can be added, removed and updated dynamically at runtime. Thus, we need to modify the hypervisor to provide an interface for management of policies. This interface is not complicated since we only need to load executable code into the hypervisor space. Our policy is written by the administrator using a C-like policy language. We do not adopt existing policy languages [9,12,17,18] since they are not sufficiently flexible and powerful. We use a customized compiling tool chain to compile the policy into executable code. Finally, the code is transferred to the hypervisor.

2.4 Policy Description Language

Our policy description language is designed to be a real "programming language" that is powerful and flexible and can be easily compiled to native code. Previous sandboxing policy languages do not fulfill this need so we do not adopt them in our system. To better illustrate our policy language, we first show a policy example written in this language in Lst. 1.

```
/* notepad.exe NtOpenFile */
sandbox_t policy(unsigned int params)
{
    sandbox_t result = permit;
    wchar_t filename1[] = L"password.txt";
    wchar_t filename2[] = L"secret.txt"
    wchar_t *filename;
    unsigned int pos, len;
    guest_read_int_at(params + 12, &pos);
    guest_read_int_at(pos + 8, &pos);
    guest_read_int_at(pos, &len);
    filename = (wchar_t *)malloc(len); /* assume malloc succeeds */
    guest_read_string_at(pos + 4, len, filename);
    if (!wstrcmp(filename, filename1))
    {
        result = disallow;
    }
    else if (!wstrcmp(filename, filename2))
    {
        guest_write_int_at(params + 4, -1);
        result = deceive;
    }
    free(filename);
    return result;
}
```

Listing 1. Policy Example

We can see from the above sample that our policy language is C-like. It can be considered as a subset of C programming language since not all features are necessary and a few limitations are imposed:

- The first line is a *directive*, i.e. a comment indicating the target program and system call name, which is similar to the `#!/bin/bash` in a shell script.
- The code cannot `#include` files since it will run in hypervisor kernel.
- Float types and computations are not supported, which are also unnecessary.
- Global variables are not allowed, and literal strings must be initialized with a `char`/`wchar_t` array, since the final binary does not have a constant section.
- The code must have the `policy` function and be self-contained. User-defined functions are allowed (though not shown here), and built-in functions can be used such as memory management (e.g., `malloc`), string operation (e.g., `strcpy`), and guest memory reading/writing functions.

3 Implementation Details

In this section, we detail our implementation with a focus on how to modify the existing VMM (hypervisor and management domain) in order to satisfy our needs. Our prototype system uses Xen hypervisor (version 4.1.2), while the dom0 (management domain) and the domU (guest domain) are CentOS 5.5 (64bit) with a patched kernel (version 2.6.34.4) and Windows XP SP3 (32bit), respectively. The development machine has an Intel Core i5 processor with the latest hardware assisted virtualization support. In the following, we present some implementation details for the key techniques in our approach.

3.1 Data Structures and Definitions

We first briefly introduce some key data structures used in our system. As is shown in Fig. 3, our policy library is implemented as a single list and its each node is a `policy_entry` corresponding to a process. Another important data structure is `policy_item`, which is used for loading the executable policy code to the hypervisor. It pertains to one process and one system call. The meanings of most fields are evident, so we do not explain them here. The following sections will explain how these data structures are made use of.

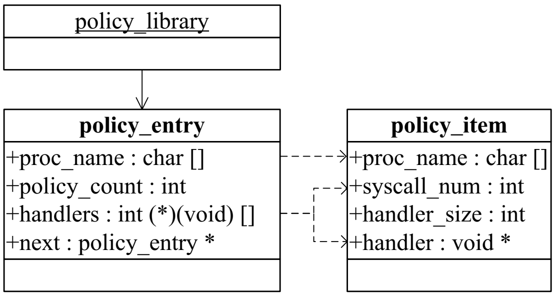


Fig. 3. Data Structures

3.2 Additional VMexit Handler

In order to capture necessary events, we need to modify some existing *VMexit handlers*. Moreover, several events must be intentionally processed to trigger *VMexit* events. We detail how these events are intercepted and handled.

Process Switch Interception. Since our policy is per process, we need to intercept process switches in order to correctly sandbox target applications. The task state segment (TSS) mechanism provided by the x86 architecture is not used by modern operating systems for task switching. In contrast, paging mechanism is widely adopted by operating systems to isolate process spaces. Hence, process switch involves the alteration of page table base address register (`cr3` under x86). Moreover, though Windows uses a thread-based scheduling method, the context switch routine will not overwrite the `cr3` register if the upcoming thread and current thread belong to the same process. Therefore, we consider VMexit events caused by `cr3` write to be process switches.

When such an event occurs, we read the process name from the kernel data structure, and then look it up in the policy library. We traverse the single list and compare the current process name with the `proc_name` field. If a match is found, the `handlers` field is copied to a per-domain variable `active_handlers`, and system call interception is enabled. Otherwise, system call interception is disabled in order to retain performance.

System Call Interception. Intercepting the system call is a pivotal step in our system. Unfortunately, neither software interrupt nor fast system call (`sysenter`) can be directly intercepted. Hence, we must deliberately trigger some events that can trap into the hypervisor. Multiple ways can be used to achieve this. After comparing their pros and cons, we take the method similar to that in Ether [4], i.e., deliberately modifying the `SYSNENTER_EIP` model specific register (MSR) to generate a page fault. We observe that no program actually uses the conventional software interrupt mechanism to perform system calls, and thus do not intercept such interrupts. Existing approaches are available if necessary[4].

To implement this, we first need to intercept access to the `SYSENTER_EIP` MSR. For write operations, we store the value at a safe place for future use, while for read operations, we always return the real value for transparency. When system call interception is enabled, a carefully chosen magic value is written to that MSR. Thus, we consider a page fault to be a system call if 1) the page fault linear address (`cr2` under x86) is equal to the magic value and 2) the page fault error code indicates an instruction fetch.

When a system call is intercepted, we look up the corresponding handler in the per-domain variable `active_handlers` using system call number (`eax` under x86). The handler is executed if it exists. Otherwise, the system call is *permitted* by default. The handler returns a value that is either *permitted*, *disallowed* or *deceived*. For *permitted* system calls, we simply assign the saved real `SYSENTER_EIP` value to the `eip` register. For *disallowed* and *deceived*, we skip the system call, and return error and success, respectively. Note that when `sysenter` is executed, the current privilege level (CPL) will be ring 0. Thus, to skip the system call, we must get back to ring 3. This is implemented by preparing several `sysexit` related registers (e.g., `ecx`, `edx`) and pointing `eip` to a `sysexit` instruction.

3.3 Management of Policies

We load policies into the hypervisor dynamically at runtime to avoid performance penalty caused by context switches between the hypervisor and the management domain. We use a technique called *runtime hypervisor manipulation*, i.e., we create a hypervisor interface and employ the *hypercall* mechanism for implementation.

Hypercall is a domain-hypervisor communication mechanism that is similar to system calls in the operating system. To make use of it, we first register a new hypercall in Xen named `do_vccbox_op`, which has only one argument, a pointer to the structure `struct policy_item`, and then implement its handler routine. If the policy with the same process name and system call number exists, we consider it to be a *policy update*, otherwise it is treated as a *policy add*. Specially, if the value of the `handler_size` field is 0, we consider it as a *policy removal*. Upon policy removal, if the policy count decreases to 0, the whole policy entry is removed from the library. A policy manager in the management domain fills the structure `struct policy_item` with necessary information and then issues a hypercall to tell the hypervisor what to do.

We observe that policy code execution and policy management are concurrent, so a race condition that the currently executing policy is being updated or removed may occur and must be eliminated. To this end, we use a *spinlock* to synchronize these two actions. We also point it out here that, though our mechanism is similar to *loadable kernel modules* under Linux, we do not consider this to be an insecure factor because, 1) the whole VMM is considered as our trusted computing based by assumption, 2) this mechanism is not designed to accommodate all loadable modules but only our policies, i.e., it is a dedicated channel for policy management, not a generic interface.

3.4 Policy Code Generation

Generating the policy code is an essential part in our system. Since our policy code ultimately runs in the hypervisor, we must keep the application binary interface compatible. Thus, we use the same arguments as Xen is compiled. Moreover, we need a preprocessing step to add necessary dependencies (e.g., declarations of guest memory reading/writing functions) to the policy file in order to make the policy compilable by *gcc*. Thus, we devise the following code generation procedure shown in Fig. 4.

The first step is to validate whether the policy text conforms to our limitations. It is performed by canonical tokenizing and parsing tools (i.e., *flex* and *bison*)[1]. Once the policy is validated, we use a preprocessing part to add some necessary declarations and definitions to the policy text. Then, the completed compilable policy is fed to *gcc* to generate an object file in ELF format using arguments obtained from Xen's *makefile*. Next, we obtain the addresses of the functions in the hypervisor that is called by our policy. For example, our guest memory reading functions are implemented via `hvm_copy_from_guest_virt_nofault`. We look up relevant information in the object file (e.g., symbol table) and then fill the corresponding locations with real addresses. This process can be considered as a simplified "linking". After that, the policy function binary can be loaded into the hypervisor using the hypercall mechanism mentioned above.

[1] This part is not yet fully implemented in our current prototype system.

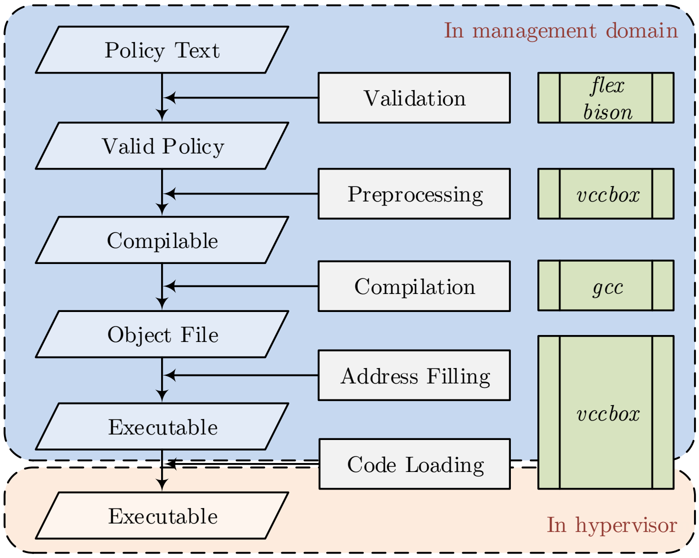


Fig. 4. Procedures of Policy Code Generation

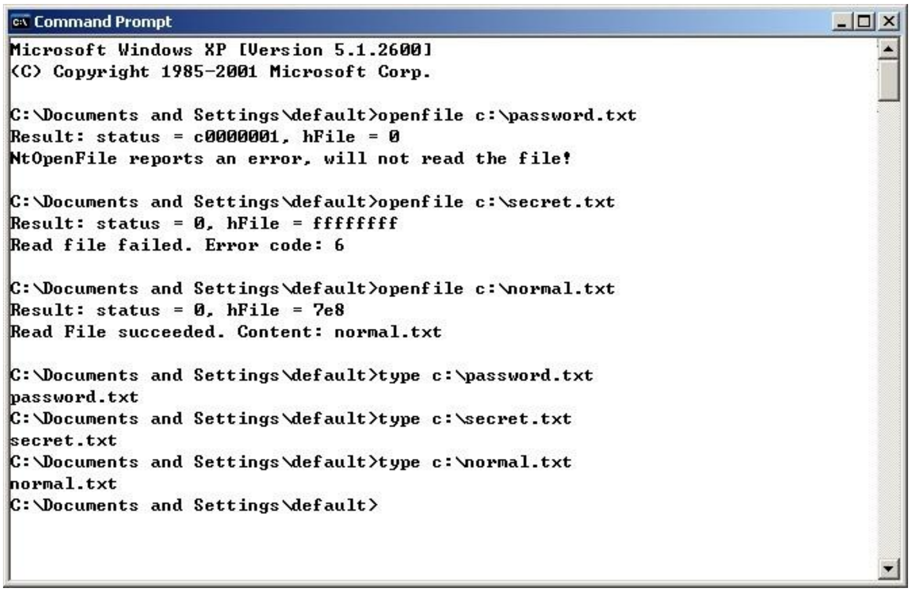
4 Evaluation

In this section, we present the analytical and experimental evaluation of our VCCBox prototype. The two goals of our evaluation are to demonstrate that VCCBox can sandbox real applications, and to measure the performance degradation introduced by our system. The following experiments were all conducted on a machine with Intel Core i5-760 processor and 8GB memory. The version of Xen used in our experiment is the latest 4.1.2 and the dom0 is 64 bit CentOS 5.5 with kernel version 2.6.34.4. The guest OS is Windows XP SP3 allocated with one processor core and 2 GB memory.

4.1 Effectiveness Evaluation

In order to evaluate the effectiveness of our system, we write three different policies targeting the same system call. We choose `NtOpenFile` here since it is representative. (Note that not all system calls can be deceived). In fact, the policy example in Lst. 1 is for this system call. We give a brief explanation of the policy here. `NtOpenFile` has 6 parameters. The first one is an output parameter used to return the handle of the opened file, and is our deception target. The third parameter is a structure, which designate the path of the file to be opened. Thus, the sample policy means: if the program tries to open “password.txt”, it will get an error; if it tries to open “secret.txt”, it will be provided with an invalid handle; otherwise, the open operation is successful.[2]

[2] Note that the sample policy is only for demonstration and still needs to be improved for practical use.



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\default>openfile c:\password.txt
Result: status = c0000001, hFile = 0
NtOpenFile reports an error, will not read the file!

C:\Documents and Settings\default>openfile c:\secret.txt
Result: status = 0, hFile = fffffff
Read file failed. Error code: 6

C:\Documents and Settings\default>openfile c:\normal.txt
Result: status = 0, hFile = 7e8
Read File succeeded. Content: normal.txt

C:\Documents and Settings\default>type c:\password.txt
password.txt
C:\Documents and Settings\default>type c:\secret.txt
secret.txt
C:\Documents and Settings\default>type c:\normal.txt
normal.txt
C:\Documents and Settings\default>
```

Fig. 5. Effectiveness Evaluation Result

We devise a test program to open the file designated in its argument. The program is sandboxed by our system, and we provide different arguments to this program and observe its output. We call `NtOpenFile` directly from `ntdll.dll` for accuracy, and print the return value and the first parameter containing the opened file handle. To verify the correctness of the handle, we use it as the parameter of `ReadFileEx`, a user space function that reads file content from a handle. Moreover, we use `type` command (similar to `cat` under Linux) to show the real file content for comparison. The result is shown in Fig. 5, from which we can see that our policy is successfully enforced. For `password.txt`, the system call is disallowed, so the return value is set to `0xC0000001`, indicating a failure. Our program checks this failure and reports it. For `secret.txt`, the system call is deceived. We set the return value to 0, meaning a successful system call, but set the output parameter of `NtOpenFile` to `0xFFFFFFFF`, which indicates `INVALID_HANDLE`. Our program reports an error code 6, which exactly means `ERROR_INVALID_HANDLE`, proving that our method has successfully *deceived* the system call[3]. Our test program is not designed to be malicious. However, due to the higher privilege and ability to intercept events of the virtual machine, our method can not be circumvented.

4.2 Performance Evaluation

The runtime overhead of our system comes from the additional VMexit handler routines and hypercalls. However, policy management is not a periodical event,

[3] A real `NtOpenFile` does not necessarily returns 0 when it provides an invalid handle. Here we only use this sample to indicate that our *deceived* policy works correctly.

and it does not often happen in practice since system administrators do not frequently change the policies once they are successfully loaded. Moreover, context switches occur much less frequently than system calls, and thus contribute little to the performance degradation according to our experiences. Therefore, we focus on the performance penalty caused by intercepting and handling system calls.

We look into the difference between normal system call execution and sandboxed execution. We denote the time for normal system call as τ_x. If an application is not sandboxed, we do not enable system call interception, hence incurring no performance overhead. If an application is sandboxed, all of its system calls will be trapped into the hypervisor. Thus, unhandled system calls will go through the process of interception *without* policy execution. This time is denoted as τ_e. While handled system calls are intercepted with policy execution, so we denote the time of policy execution as τ_p. Note that since only one policy for a program can lead to system call interception, and the per system call policies will not run together for one system call, the number of policies does not influence the overall performance. Tab. 1 shows the time of execution for different situations.

Table 1. Time Comparison for Different Situations

Situation	Time
Normal	τ_x
Intercept + No handler	$\tau_e+\tau_x$
Intercept + Handler + *permitted*	$\tau_e+\tau_p+\tau_x$
Intercept + Handler + *disallowed/deceived*	$\tau_e+\tau_p$

However, the magnitude of τ_e, τ_p and τ_x is indeterminate, since τ_p and τ_x are respectively per policy and per system call, and the time of execution depends on which path is taken. Thus, we cannot theoretically calculate the performance impact, and micro-benchmark is rendered difficult. So we perform macro-benchmark to measure the performance impact caused by our system. Thus, we enable system call interception for all processes and *permit* all system calls.

Results from Benchmark Tools. We use Super PI, DAMN Hash Calculator, Everest, Crystal DiskMark to benchmark the performance overhead of the processor, memory access and I/O introduced by our system. The results are summarized in Fig. 6, which exhibit that our system is highly efficient.

Results with Real Workload Experiment. To evaluation the efficiency of VCCBox under real workloads, we employ kernel building, a comprehensive task that represents a typical workload and is used widely for profiling. Specifically, we record the time of building the Windows Research Kernel (WRK) under different circumstances. Through scrutinizing possible policies, we observe that the most time-consuming part of a policy is read memory access operations, since the

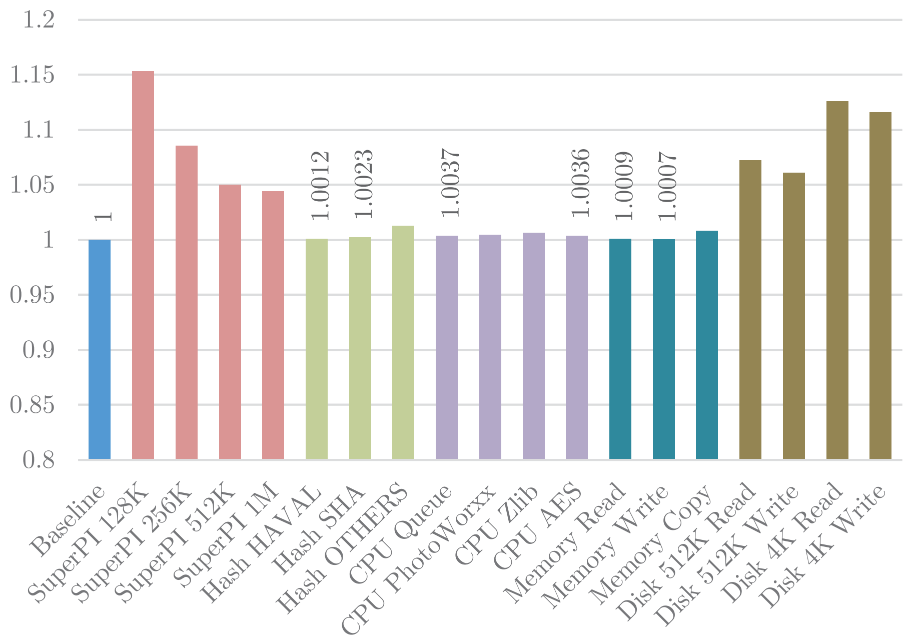


Fig. 6. Benchmark results (Lower is better)

kernel data structures are complicated and frequently point to other structures, leading to multiple memory accesses in retrieving a single piece of information. For example, to obtain the filename from the handle, a parameter of `NtReadFile`, around 7 memory reads are performed. Such read access to memory content is implemented by `hvm_copy_from_guest_virt_nofault` function (`HVMCOPY` for short). Thus, we add different numbers of synthetic `HVMCOPY` function calls in the system call handler and record the kernel building time. The result of the experiment is shown in Tab. 2.

Table 2. Time for Kernel Building

Situation	Time (ms)	Overhead
Normal	99734	N/A
Intercept + No `HVMCOPY`	102750	3.0%
Intercept + 10 `HVMCOPY`s	106617	6.9%
Intercept + 20 `HVMCOPY`s	110595	10.9%
Intercept + 30 `HVMCOPY`s	114094	14.4%
Intercept + 40 `HVMCOPY`s	119922	20.2%

We can see the system call interception itself causes only 3.0% runtime overhead. Moreover, the performance degradation is gradually aggravated as the number of `HVMCOPY`s (representing the policy complexity) increases. Ordinary policies such as preventing a file with specific name from being read will never use more than 40 `HVMCOPY`s, and two decisions (*disallowed* and *deceived*) will cause the system calls to be skipped. Thus, we may safely consider the overhead of 20.2% as the worst case, which is acceptable.

5 Discussion and Future Work

The above evaluation exhibits that our current prototype can work effectively and efficiently. In this section, we discuss several limitations of our VCCBox prototype and some future work.

5.1 Extensibility Related Issues

VMM and OS Support. Our current implementation targets Xen and Windows only. However, other hypervisors and operating systems can also be supported. KVM [13] is a rising star among VMMs and wins high favor from both academia and industry, while Linux is the dominating operating system in production environment. We would like to integrate support for them into future versions of VCCBox.

Heterogeneous VM Situation. We currently assume that all virtual machines run the same operating system and/or software. The server environment for load balancing usually satisfies this requirement. However, virtual machines in multi-tenant cloud environment can be *heterogeneous*. Fortunately, our architecture can support such situation. Preliminarily, we can add VMID and OS-Type entry to the policy to distinguish virtual machines and guest operating systems, Further, we can automatically identify the running operating system using oracles [14] or fingerprinting methods [10].

5.2 Insufficiencies and Improvement of Policy

Policy Complexity. Our policy description language is flexible and powerful, which sacrifices its simplicity. It is more complex than other current policy languages and requires the writer (usually a system administrator) to be familiar with operating system data structures. We intend to reduce the policy complexity by integrating recent advances in automatically narrowing the semantic gap such as Virtuoso [5] and VMST [6].

Policy Robustness. Since we need to retrieve the data inside the virtual machine, our policies depend heavily on virtual machine introspection [8]. However, this mechanism can be subverted by direct kernel structure manipulation (DKSM), a technique that directly modified the kernel data structures to mislead security applications [2]. This issue is considered as an open problem by state-of-the-art tools on bridging the semantic gap [5,6] and is not solved to date. Fortunately, DKSM can be prevented by our sandbox mechanism by disallowing untrusted drivers to be loaded, since DKSM requires the kernel privilege to work. In the future, we plan to investigate reliable virtual machine introspection method so as to thoroughly address this issue.

Policy Debugging. Since our policies are ultimately running as code inside the hypervisor, a bug such as access violation can cause more severe results such as crashing the whole physical machine. Hence, we need a mechanism to debug the policy binary. To this end, we can add a *debug* option in our preprocessing module. When this option is enabled, we insert several validations into the generated compilable C code and use dedicated secure versions of necessary functions. The debug version of C code is not designed for daily use since the additional security examination will degrade the performance. The primary purpose of policy debugging is to prevent the policy writer from making careless mistakes.

6 Related Work

6.1 Hypervisor Based Security

With the advent of the cloud computing era, virtualization technology has been widely adopted in research of system security. Hardware assisted virtualization provides powerful processor-level support for privilege separation, memory isolation and access control, which are all desirable features for security applications. Currently, security research efforts based on hardware assisted virtualization can be categorized in malware analysis [4,15,25], kernel protection [19,22,24] and execution monitoring [11,16,20].

Malware analysis platforms use hardware assisted virtualization chiefly for the purpose of transparency. In essence, malware analysis tools have different goals with sandboxes. They *passively observe* the behavior of potentially harmful programs, while sandboxes *actively interfere* with the execution process of untrusted applications. Some techniques of malware analysis can be naturally used for sandboxing. For example, the system call interception method adopted by our system is first proposed in Ether [4].

Kernel protection mechanisms do not address the issue of application behavior confinement. Instead, they are concerned about how to secure the operating system kernel. Thus, most of them fall into the category of anti-rootkit mechanism. In contrast, VCCBox confines the user-level behaviors of an application, and does not care about kernel execution. Interestingly, VCCBox can defeat kernel-level rootkit in a trivial way, i.e.,enforcing a policy to prevent untrusted applications from loading malicious kernel extensions via system calls.

Execution monitoring is a fundamental underlying technique for malware analysis and other security tasks. A significant challenge to correctly monitoring the process inside the virtual machine is *semantic gap* [3], which can be chiefly addressed by virtual machine introspection. However, some execution monitoring tools employs a *hybrid* approach, i.e., using an in-guest component to actually monitor the execution process, while the hypervisor protects this component from being detected or tampered. Though this hybrid approach better solves semantic gap and improves efficiency, VCCBox insists the conventional *out-of-the-box* approach in order not to lose the ease-of-deployment.

6.2 Application Sandbox

Application sandboxing is not new in security area. Many approaches have been used to construct sandboxes. Here we introduce a few representative relevant research efforts in sandbox and compare them with our VCCBox.

Janus [9] is an early and simple sandbox system. It runs entirely in user space and takes advantage of the */proc* interface under Linux for system call interposition. Hence, it is subject to race condition attacks such as "Time of Check to Time of Use" (TOCTTOU), and is liable to be bypassed [7]. Moreover, this interface is not available under such operating systems as Microsoft Windows.

Systrace [17] takes a hybrid approach that involve both user-space and kernel-space to address the TOCTTOU race condition. Systrace however uses an interactive policy generator, which makes it not suitable for production environment where nobody can always stay before the screen. Systrace also assumes the sandboxed application is *intrinsically benign* and only behaves viciously under external attacks, which limits it usage to defend *intrinsically malicious* applications.

Authenticated system calls [18] is a cryptographic approach towards securing the system calls. To initialize, the to-be-sandboxed application is processed by a *trusted installer*, which statically analyzes the program to locate system calls and mine their legal usages to generate policies. Then, it replaces each conventional system call with an authenticated one containing the policy and a message authentication code (MAC). The kernel verifies the MAC and enforces the policy when executing the system call. A main shortcoming of the sandbox is that when the program is obfuscated, the static analysis can hardly get useful system call information.

TxBox [12] introduces the concept of transaction from database for sandboxing. It allows a program to run as a transaction, hence is able to roll back any devastating impact caused by the program. The transaction mechanism has several inherent advantages when used to construct a sandbox. For example, it allows the concurrent execution of sandboxed application and damage detection process, and is able to recover from a multi-staged attack. However, TxBox relies on a dedicated Linux kernel with transaction features, limiting its practicality.

7 Conclusion

In this paper, we have presented the motivation, design, implementation and evaluation of VCCBox, a hypervisor-based sandbox which eliminates various deficiencies of previous work and is a practical sandbox solution for cloud environment. In particular, by leveraging the state-of-the-art hardware assisted virtualization and implementing the sandbox routine totally inside the VMM, VCCBox can not be bypassed and is easy to deploy in virtual cloud infrastructure. Moreover, *runtime hypervisor manipulation* is adopted to dynamically load policies into the hypervisor, which ensures the high performance of VCCBox.

References

1. Azab, A.M., Ning, P., Wang, Z., Jiang, X., Zhang, X., Skalsky, N.C.: HyperSentry: enabling stealthy in-context measurement of hypervisor integrity. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, pp. 38–49. ACM, New York (2010)
2. Bahram, S., Jiang, X., Wang, Z., Grace, M., Li, J., Srinivasan, D., Rhee, J., Xu, D.: DKSM: subverting virtual machine introspection for fun and profit. In: Proceedings of the 29th IEEE Symposium on Reliable Distributed Systems, SRDS 2010, pp. 82–91. IEEE Computer Society, Washington, DC (2010)
3. Chen, P.M., Noble, B.D.: When virtual is better than real. In: Proceedings of the 8th USENIX Workshop on Hot Topics in Operating Systems, HotOS 2001, pp. 133–138. IEEE Computer Society, Washington, DC (2001)
4. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: malware analysis via hardware virtualization extensions. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS 2008, pp. 51–62. ACM, New York (2008)
5. Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J., Lee, W.: Virtuoso: narrowing the semantic gap in virtual machine introspection. In: Proceedings of the 32nd IEEE Symposium on Security and Privacy, S&P 2011, pp. 297–312. IEEE Computer Society, Washington, DC (2011)
6. Fu, Y., Lin, Z.: Space traveling across VM: automatically bridging the semantic-gap in virtual machine introspection via online kernel data redirection. In: Proceedings of the 33rd IEEE Symposium on Security and Privacy, S&P 2012, San Francisco, CA (May 2012)
7. Garfinkel, T.: Traps and pitfalls: practical problems in system call interposition based security tools. In: Proceedings of the 10th Annual Network and Distributed Systems Security Symposium, NDSS 2003 (2003)
8. Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: Proceedings of the 10th Annual Network and Distributed Systems Security Symposium, NDSS 2003 (2003)
9. Goldberg, I., Wagner, D., Thomas, R., Brewer, E.A.: A secure environment for untrusted helper applications. In: Proceedings of the 6th USENIX Security Symposium, Security 1996. USENIX Association, Berkeley (1996)
10. Gu, Y., Fu, Y., Prakash, A., Lin, Z., Yin, H.: OS-Sommelier: memory-only operating system fingerprinting in the cloud. In: Proceedings of the Third ACM Symposium on Cloud Computing, SoCC 2012, pp. 5:1–5:13. ACM, New York (2012)
11. Gu, Z., Deng, Z., Xu, D., Jiang, X.: Process implanting: a new active introspection framework for virtualization. In: Proceedings of the 30th IEEE International Symposium on Reliable Distributed Systems, SRDS 2011, pp. 147–156. IEEE Computer Society, Washington, DC (2011)
12. Jana, S., Porter, D.E., Shmatikov, V.: TxBox: building secure, efficient sandboxes with system transactions. In: Proceedings of the 32nd IEEE Symposium on Security and Privacy, S&P 2011, pp. 329–344. IEEE Computer Society, Washington, DC (2011)
13. Kivity, A., Kamay, Y., Laor, D., Lublin, U., Liguori, A.: KVM: the Linux virtual machine monitor. In: Proceedings of the 9th Ottawa Linux Symposium, vol. 1, pp. 225–230 (2007)
14. Litty, L., Lagar-Cavilla, H.A., Lie, D.: Hypervisor support for identifying covertly executing binaries. In: Proceedings of the 17th USENIX Security Symposium, Security 2008, pp. 243–258. USENIX Association, Berkeley (2008)

15. Nguyen, A.M., Schear, N., Jung, H., Godiyal, A., King, S.T., Nguyen, H.D.: MAVMM: lightweight and purpose built VMM for malware analysis. In: Proceedings of the 25th Annual Computer Security Applications Conference, ACSAC 2009, pp. 441–450. IEEE Computer Society, Washington, DC (2009)
16. Payne, B.D., Carbone, M., Sharif, M., Lee, W.: Lares: an architecture for secure active monitoring using virtualization. In: Proceedings of the 29th IEEE Symposium on Security and Privacy, S&P 2008, pp. 233–247. IEEE Computer Society, Washington, DC (2008)
17. Provos, N.: Improving host security with system call policies. In: Proceedings of the 12th USENIX Security Symposium, Security 2003. USENIX Association, Berkeley (2003)
18. Rajagopalan, M., Hiltunen, M., Jim, T., Schlichting, R.: System call monitoring using authenticated system calls. IEEE Transactions on Dependable and Secure Computing 3(3), 216–229 (2006)
19. Seshadri, A., Luk, M., Qu, N., Perrig, A.: SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In: Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles, SOSP 2007, pp. 335–350. ACM, New York (2007)
20. Sharif, M.I., Lee, W., Cui, W., Lanzi, A.: Secure in-VM monitoring using hardware virtualization. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS 2009, pp. 477–487. ACM, New York (2009)
21. Wang, Z., Jiang, X.: HyperSafe: a lightweight approach to provide lifetime hypervisor control-flow integrity. In: Proceedings of 31st IEEE Symposium on Security and Privacy, S&P 2010, pp. 380–395. IEEE Computer Society, Washington, DC (2010)
22. Wang, Z., Jiang, X., Cui, W., Ning, P.: Countering kernel rootkits with lightweight hook protection. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS 2009, pp. 545–554. ACM, New York (2009)
23. Wang, Z., Wu, C., Grace, M., Jiang, X.: Isolating commodity hosted hypervisors with HyperLock. In: Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys 2012, pp. 127–140. ACM, New York (2012)
24. Xiong, X., Tian, D., Liu, P.: Practical protection of kernel integrity for commodity OS from untrusted extensions. In: Proceedings of the 18th Annual Network and Distributed System Security Symposium, NDSS 2011 (2011)
25. Yan, L.-K., Jayachandra, M., Zhang, M., Yin, H.: V2E: combining hardware virtualization and software emulation for transparent and extensible malware analysis. In: Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments, VEE 2012, pp. 227–238. ACM, New York (2012)