Load-based Covert Channels between Xen Virtual Machines

Keisuke Okamura
Department of Computer Science,
The University of Electro-Communications
1-5-1 Chofugaoka, Chofu-shi, Tokyo, Japan
okamura@ol.cs.uec.ac.jp

Yoshihiro Oyama
Department of Computer Science,
The University of Electro-Communications
1-5-1 Chofugaoka, Chofu-shi, Tokyo, Japan
oyama@cs.uec.ac.jp

ABSTRACT

Multiple virtual machines on a single virtual machine monitor are isolated from each other. A malicious user on one virtual machine usually cannot relay secret data to other virtual machines without using explicit communication media such as shared files or a network. However, this isolation is threatened by communication in which CPU load is used as a covert channel. Unfortunately, this threat has not been fully understood or evaluated. In this study, we quantitatively evaluate the threat of CPU-based covert channels between virtual machines on the Xen hypervisor. We have developed CCCV, a system that creates a covert channel and communicates data secretly using CPU loads. CCCV consists of two user processes, a sender and a receiver. The sender runs on one virtual machine, and the receiver runs on another virtual machine on the same hypervisor. We measured the bandwidth and communication accuracy of the covert channel. CCCV communicated 64-bit data with a 100% success rate in an ideal environment, and with a success rate of over 90% in an environment where Web servers are processing requests on other virtual machines.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Measurement

Keywords

Covert channels, virtual machine monitors

1. INTRODUCTION

Virtual hosting has become an essential service in today's information industries. In particular, much attention has been paid to hosting services that are built on a virtual machine monitor (VMM). These services partition resources on a physical machine into multiple virtual machines and rent each virtual machine to end users.

A major advantage of VMM-based virtual hosting is *strong isolation*. Intuitively, a VMM gives users an illusion of monopolizing a completely set up computer. VMM-based hosting provides each user with a set of virtual hardware that is strongly isolated from that of others. In this type of hosting, distinct instances of operating systems are run in distinct environments (virtual machines), and hence, logical resources such as files are not shared among multiple environments. That isolation feature is strongly required when hosting the environments of mutually untrusting users.

VMM-based hosting is particularly attractive for those who would like to store and manipulate secret data in a rented environment, because multiple virtual environments ordinarily do not share any logical resources. Therefore, information leakage can basically be prevented by monitoring network communication, as in the case of an environment built on a dedicated physical machine. If a malicious program on one virtual machine attempts to send secret data to an external environment (including another virtual machine hosted on the same physical machine), network monitoring software such as [12, 15, 16, 17] can easily detect and prevent the communication.

Unfortunately, as has been pointed out earlier [7, 9], conspiring programs can communicate with each other through a channel that is not intended for communication by administrators. The channel is often called a *covert channel*. A wide range of resources are known to be usable as covert channels, e.g., TCP headers [2], timing of computation [18], and memory access latency [3, 4, 10]. A monitoring system that examines only ordinary channels fails to notice information leakage through covert channels. In VMM-based hosting, malicious programs can create covert channels between different virtual machines [11].

One of the most convenient resources for covert channels between virtual machines is CPU load, which can be approximated by the amount of time taken for certain computations. Since CPUs or cores are often shared among different virtual machines and processes, a malicious process (e.g., spyware) on one virtual machine can secretly communicate with its peer on another virtual machine by changing the CPU load and making the peer recognize the change. Ordinary security systems cannot detect or prevent the communication. CPU load is a convenient resource for attackers because changing the CPU load does not require administrator privileges, and it is easier to control in a time-sensitive manner than most other indicators, such as page fault rates.

SAC'10 March 22-26, 2010, Sierre, Switzerland.

It is well known that multiple processes on a single operating system can communicate with each other using the CPU load. An important aspect yet to be investigated is how fast and how accurately one can communicate through a CPU-load-based covert channel across the boundary between virtual machines. The threat of a covert channel depends on its bandwidth and accuracy. As per our information, no work has quantitatively shown the attainable bandwidth and accuracy of inter-virtual-machine covert channels based on the CPU load.

In this work, we develop CCCV (Covert Channels using CPU loads between Virtual machines), a covert channel communication system between virtual machines on the Xen hypervisor [1]. It enables processes in DomU virtual machines to communicate with each other using a physical CPU as the communication medium. CCCV makes processes on different virtual machines communicate through information on the execution times of small program code. We quantitatively evaluate the ability of CCCV; in particular, we clarify its bandwidth and accuracy. It is implemented without modifying the code of the Xen hypervisor or a guest OS kernel. Experimental results show that CCCV processes can communicate at 0.49 bps with 100% accuracy in an ideal environment and at the same bps with about 91% accuracy even when other virtual machines are hosting a moderately loaded Web server on the same hypervisor.

This study focuses on CPU load and does not consider the creation of covert channels with other resources such as memory access latency in this work. We can improve the bandwidth and accuracy of CCCV by combining CPU load (execution time) information with information about other resources. However, this is left for future work, and here we concentrate on identifying a lower bound of the bandwidth and accuracy of a covert channel system that controls and observes execution time information alone.

The primary target situation for the current CCCV implementation is one in which different virtual machines share a physical CPU (core). This condition is always satisfied on a single-core uniprocessor machine and sometimes satisfied on multi-core machines. This paper describes a basic idea for extending CCCV to work well also when different virtual machines are not always mapped to the same set of cores. This paper further shows preliminary experimental results for evaluating the extension. Before rushing into collecting experimental results in complex conditions, however, we focused on obtaining results in the simple situation above. The results will be useful as fundamental data for future investigation.

This paper is organized as follows. Section 2 describes CPU schedulers in Xen, and Section 3 explains the implementation of the proposed system, CCCV. Section 4 shows our experimental results, and Section 5 compares our work with related work. Section 6 concludes this paper with a brief summary and the description of several future projects.

2. CPU SCHEDULING IN XEN

Virtual machines on the Xen hypervisor are called *domains*. The hypervisor partitions and distributes the cycles of a physical CPU to domains. The hypervisor hides physical CPUs from virtual machine users, and instead, shows virtual CPUs (VCPUs) to each domain. Guest OS code controls assigned VCPUs, while the *domain scheduler* in the Xen hypervisor controls mapping between VCPUs and physical CPUs. Naturally, the hypervisor can map multiple VCPUs onto one physical CPU. The domain scheduler assigns a portion of the physical CPU time to each VCPU, and a process scheduler in a guest OS kernel further distributes the assigned CPU time to processes. A domain pauses when no physical CPU is available for the VCPUs assigned to the domain. Hence, the amount of time that elapses during the execution of some computations includes the pause time of the underlying domain. CCCV uses this characteristic to create a covert channel.

Two domain schedulers are available in the current version of Xen: the SEDF scheduler and the credit scheduler. Since the credit scheduler is standard in recent versions, CCCV presumes its use.

The credit scheduler periodically assigns *credits* (CPU time allowed for use) to each VCPU. The CPU time actually consumed by a VCPU is subtracted from its credits. The administrator assigns a *weight* and a *cap* to each domain, and the credit scheduler determines the credits given to each VCPU according to these two values. Intuitively, the weight represents the priority of a domain. Credits are usually distributed in proportion to the weight of the domains. The cap represents an upper limit on the CPU time consumed by a VCPU. The default value of a cap is zero, which means that there is no limit on CPU time. The state of a VCPU that has consumed all its credits is called *over*, and the other state is called *under*. The credit scheduler schedules VCPUs with the under state in a round-robin manner. VCPUs with the over state receive CPU time only when all VCPUs with the under state are idle.

3. PROPOSED SYSTEM

3.1 Threat Scenario

We assume that an attacker has a way to inject spyware into a domain managed by another person. A guest OS in the domain stores secrets such as passwords or credit card numbers. We call this domain a sender domain. The person or the VMM administrator intends to prevent information leakage by introducing network monitoring software (e.g., [12, 15, 16, 17]) or a well-configured firewall in the domain, or by building a virtual private network that isolates the domain from the Internet. Spyware in the domain cannot leak secrets simply by sending them through the Internet because the monitoring software may detect the leakage or the firewall or virtual private network may prevent the spyware from accessing the Internet.

We also assume that the distributor of the spyware can create or rent a domain beside the sender domain on the same hypervisor. This is sometimes possible simply by using the virtual hosting service provider that hosts the sender domain. We call this domain a receiver domain. Since the attacker manages the receiver domain, the attacker does not install any security software or firewall in this domain. The attacker executes CCCV in the sender and receiver domains (the attacker makes the spyware contain the sender part of CCCV). The spyware sends secrets to the receiver domain through a covert channel, and a malicious program in the receiver domain sends the secrets to the attacker's site via the Internet.

3.2 CPU Load as a Covert Channel

We describe the basic idea of communication by CCCV

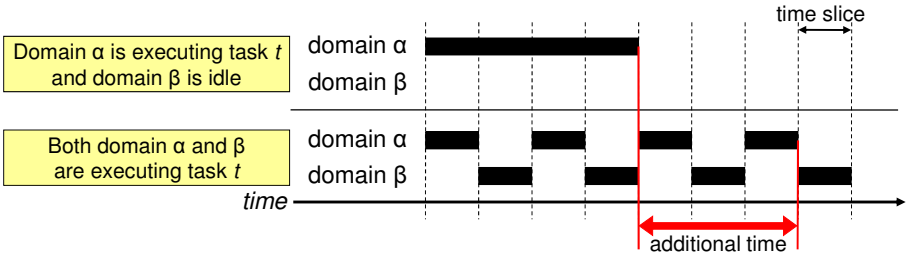

Figure 1: Change in execution time due to sharing of a physical CPU

with an example. We assume that a hypervisor is hosting two DomUs, domain α and domain β, and that VCPUs in the domains are mapped to the same CPU (core). Both domains are given the same weight and a cap of zero. CCCV runs a receiver process in domain α and a sender process in domain β. Each process executes task t in its domain. We also assume that t requires four time slices for completion, and no CPU time is consumed by programs other than t.

Figure 1 illustrates the timeline of execution in domain α in different cases. The upper part shows the case in which domain β has no runnable task. In that case, domain α consumes most of the physical CPU's time. Consequently, the amount of time that elapses during the execution of t equals four time slices. In contrast, the lower part shows the case in which domain α and domain β start executing t at the same time. Since the domains run on one physical CPU, either of the domains is alternately scheduled in every time slice. One domain must pause when the other is scheduled. Hence, the amount of time that elapses during the execution of t in domain α increases to seven time slices. This means that domain α can know whether domain β is executing some computation by measuring the time required to complete t; domain β can send information to domain α by changing the CPU load according to a communication protocol, such as "bit 1 is being sent if domain β is executing t, and bit 0 is being sent otherwise."

3.3 Overview

CCCV consists of a sender process and a receiver process. Figure 2 shows the structure of CCCV. A sender process runs in a domain from which a malicious program sends secrets. A receiver process runs in a domain in which a peer of the malicious program receives the secret. The processes run with a normal user's privilege. The sender sends secrets by changing the CPU load, whereas the receiver receives them by recognizing the changes. In one communication cycle, CCCV communicates 64 bits by repeating 1-bit communication 64 times.

Note that a 64-bit number can represent a wide range of security-critical information. An ordinary credit card number is encoded by 16 digits, which fit into 64 bits. An eight-letter UNIX password can also be encoded by 64 bits.

We describe a scheme of 1-bit communication. The sender and receiver use a common program that simply consumes CPU cycles by executing a small fixed amount of computation. Execution times of the program are used as an indicator of CPU load. We call the program *unitcomp*.

When a sender sends information to a receiver on the same hypervisor, it first makes the receiver synchronize with itself. Details of the synchronization are given in Section 3.4.1. After synchronization, the sender and receiver communicate information one bit at a time. The receiver receives a bit by measuring how often it can execute unitcomp during a standard timespan. If the sender sends bit 1, it repeatedly executes unitcomp during a standard timespan. Otherwise, it does not execute any program throughout the timespan. When a standard timespan has passed, the receiver compares the number of executions of unitcomp with a precalculated standard number. If the difference between them is larger than a threshold, the receiver receives bit 1. Otherwise, it receives bit 0.

Naturally, CPU loads in other domains can disturb communication by CCCV. We do not claim that CCCV can always achieve perfect communication. We only claim that CCCV works well if the CPU loads of processes other than the sender and receiver do not vary significantly during communication.

3.4 Communication Protocol

We describe the communication protocol of CCCV. The sender and receiver need to execute unitcomp in a synchronized manner. Therefore, both processes must synchronize the timing of communication before sending or receiving a bit stream that represents secrets. Moreover, they must pause periodically to prevent the underlying CPU from being judged as over by the credit scheduler. The current communication protocol takes these requirements into account. It consists of three phases: the synchronization phase, the confirmation phase, and the bit transmission phase.

It takes 131.5 s to communicate 64 bits in the current implementation of CCCV. The bandwidth is $64/131.5 = 0.49$ bits per s if no error occurs. The breakdown of the time is shown below.

- Synchronization phase: 2.5 s
- Confirmation phase: 1 s
- Pause between confirmation phase and bit transmission phase: 1 s
- Transmission of bits: 64 s
- Pauses between transmission of each bits: 63 s

Before starting communication, the receiver executes an initialization procedure. We show a simplified receiver algorithm in Figure 3. Lines 1–6 describe the initialization procedure executed before communication. Here, a receiver repeats an execution of unitcomp 100 times, which takes

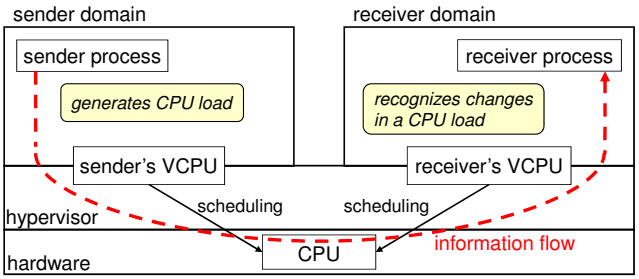

Figure 2: Structure of CCCV

about 0.1 s. After that, the receiver calculates the average time taken for one execution of unitcomp. This initialization procedure is needed to minimize the effect of CPU usage by non-communicating processes.

3.4.1 Synchronization Phase

Figure 4 shows the flow of the communication protocol of CCCV. The roots of the arrows indicate the time at which an execution of unitcomp starts, and the tips of the arrows indicate the completion time.

A sender and receiver synchronize as follows. First, the sender creates a specific CPU load pattern that indicates a communication request. A communication request is represented by executions of unitcomp that last for 1.5 s. The receiver waits for a communication request by repeating an execution of unitcomp that takes approximately 0.1 s. When a receiver detects a rise in CPU load (i.e., growth in the execution time), it starts measuring the amount of time that elapses from the rise in CPU load until the fall in the load. If the time is close to 1.5 s, the receiver judges that the sender is transmitting a communication request.

Lines 8–25 in Figure 3 show the algorithm of the receiver process in the synchronization phase. A receiver recognizes a communication request when the following conditions are satisfied:

- The receiver observes a period of time in which all execution times of unitcomp are more than the average execution time plus a certain threshold. We set the threshold to 0.03 s in the current implementation.
- The length of the period is about 1.5 s.

After a high CPU load period indicating a communication request, the sender and receiver sleep for one second. The sleep indicates the end of a communication request and adjusts the timing of the following operation.

3.4.2 Confirmation Phase

The communication confirmation phase starts after the sleep. In this phase, the receiver confirms that the observed rise in CPU load is actually caused by a communication request from the sender and not by the computation load due to other processes. In this phase, the sender pauses for a certain period of time, and the receiver calculates how often it can execute unitcomp during this period. The number is called *standard number*, and the period is called *standard timespan*. Currently, the standard timespan is 1 s. A standard number is used in the bit transmission phase. If the receiver detects a rise in CPU load during the period (i.e., if

```
1: { initialization procedure begin }
2: do 100 times
3:   measure the execution time of unitcomp
4: end
5: ave = average of execution times above
6: { initialization procedure end }
7:
8: { synchronization phase begin }
9: do infinitely
10:   start_time = current time
11:   measure the execution time of unitcomp
12:   if the execution time ≥ ave + 0.03 then
13:     repeat
14:       measure the execution time of unitcomp
15:       end_time = current time
16:     until the execution time < ave + 0.03
17:     if | end_time − start_time − 1.5 | < ϵ then
18:       { a higher CPU load observed for about 1.5 s }
19:       { the CPU load is judged as sender's }
20:       break;
21:     else
22:       { the CPU load is not judged as sender's }
23:       continue;
24: end
25: { synchronization phase end }
26:
27: { confirmation phase begin }
28: ...
```

Figure 3: Algorithm of the receiver process in the initialization procedure and synchronization phase

the standard number is larger than a precalculated average by a threshold), it judges that it received a false communication request in the synchronization phase due to CPU usage by processes other than the sender. In that case, the receiver returns to the start of the synchronization phase.

The sender and receiver then pause for another 1 s to set the state of the underlying VCPUs as under.

Here, a slight gap may still exist between the timing of the sender and receiver. The length of the gap depends on the execution time of unitcomp used in the synchronization phase. We minimize the effect of the gap by providing sufficient time for communication of each bit in the bit transmission phase.

To sum up, the CCCV protocol requires a sender to make a specific CPU load pattern before transmitting bits of useful information. This contributes to synchronization of communication and exclusion of false communication and communication attempts in error-prone conditions.

3.4.3 Bit Transmission Phase

Next, the sender and receiver start the bit communication phase, in which they communicate 64-bit information. The protocol consumes a standard timespan to communicate one bit. The receiver measures the number of executions of unitcomp during the standard timespan. If the number is less than 90% of the standard number, it receives bit 1. Otherwise, it receives bit 0.

The sender and receiver sleep for 1 s between transmissions of each bit. This sleep is introduced to prevent the underlying VCPU from being judged as over. If there was no sleep, a receiver would continuously consume CPU time, and consequently the state of the underlying VCPU would become over. As described above, a smaller amount of CPU time is assigned to an over VCPU. Let us consider the case

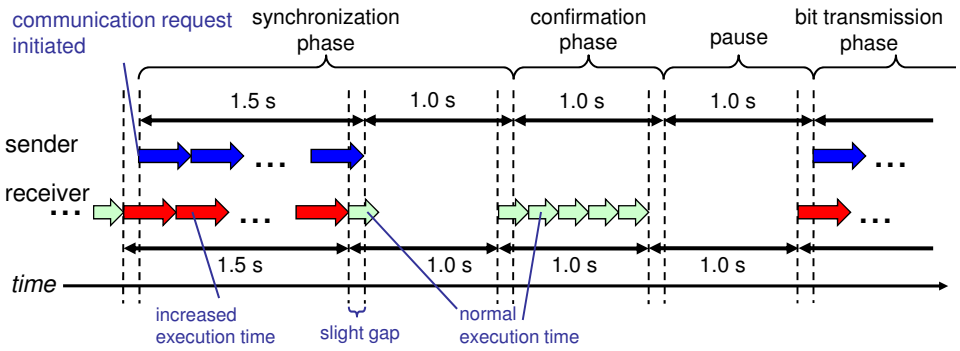

Figure 4: Flow of the CCCV communication protocol

in which the VCPU of a receiver becomes over, and a third-party process starts executing some computations. In that case, a large part of the CPU time is assigned to the third-party process, and the number of executions of unitcomp in the receiver domain decreases significantly, causing an error in the communicated bits.

No synchronization is performed in the bit transmission phase. In this phase, the communication of each bit takes a fixed amount of time (1 s), and the number of transmitted bits is also fixed to a small number, 64. Hence, the timing gap is likely to be kept small during one 64-bit communication.

When communication of bits finishes, the sender and receiver simply finish the communication without any special operation. If the processes attempt to communicate more information, they restart from the synchronization phase.

3.5 Extension to Multi-core Machines

We describe a basic idea for extending CCCV to work well also when processes in communicating domains can be mapped to different physical CPU cores. The extended protocol does not assume that VCPUs in communication domains are always mapped to the same core. In the extended protocol, the sender and the receiver create as many processes as the number of VCPUs assigned to a domain. We call the processes a sender group and a receiver group.

All processes in the sender group execute unitcomp in a synchronized manner and measure the elapsed time according to the original protocol. Each process in the receiver group also executes a receive operation in the same way as the original protocol. After 64-bit communication, however, a process in the receiver group attempts to share the bits with other processes that start a receive operation with the same timing. Then, the processes determine one bit by one bit using a majority algorithm. Bits are successfully transmitted in the extended protocol if at least one sender process and one receiver process share a physical core during communication. Processes in a sender or receiver group synchronize with each other using signals and shared memory.

4. EVALUATION

4.1 Platform

We evaluated the bandwidth and accuracy of covert channel communication by CCCV. The platform was a desktop personal computer with a Celeron 2.93 GHz and 1 GB main memory. The version of the Xen hypervisor was 3.1.4. We ran Debian etch Linux in each DomU and allocated 128 MB virtual memory to each DomU. We set the caps of all domains to zero.

4.2 Accuracy

We measured the accuracy under various conditions by changing the relative priorities of communicating domains and the CPU loads in other domains that were not related to CCCV (called third-party domains below). We attempted 64-bit communication 100 times for each condition and examined whether synchronization was successful and whether the communicated bits were correct.

4.2.1 When Other Domains are Inactive

In the first experiment, we measured the accuracy obtained when CPU loads due to processes other than the sender and receiver are extremely low. Domains hosted on the hypervisor in this experiment were Dom0, a sender domain, and a receiver domain. A minimum set of processes were running in Dom0. The results indicate communication accuracy in an ideal case. Even in this condition, Dom0 consumed approximately 4% of CPU time.

Table 1 shows the result. We varied the weights of the sender and receiver domains. S:R in the table means the ratio between the sender and receiver weights. Success indicates the number of attempts in which the communicated 64 bits were all correct. Sync failure indicates the number of failures in the synchronization phase (i.e., the receiver could not recognize the sender's communication request). Bit error indicates the number of cases in which synchronization was successful but some communicated bits were incorrect.

S:R = 1:1 is the most ideal condition, where CCCV achieved 100% accuracy. No error was observed in communicated bits in this experiment. This result suggests a characteristic of CCCV under ideal conditions: if the sender and receiver successfully synchronize, the accuracy in communicated bits is significantly high.

Unfortunately, synchronization failed when S:R was not 1:1. When the ratio between weights was increased, the accuracy decreased accordingly. The decrease in the accuracy was prominent when the receiver domain was given a larger weight.

The reason for synchronization failures changes according to the ratio between weights. When the receiver domain

Table 1: Accuracy of communication measured on a single-core machine when no third-party domain was hosted.

S:R	3:1	2.5:1	2:1	1.5:1	1:1	1:1.5	1:2	1:2.5	1:3
Success	92	98	100	98	100	61	54	16	4
Sync failure	8	2	0	2	0	39	46	84	96
Bit error	0	0	0	0	0	0	0	0	0

Table 2: Accuracy of communication measured on a multi-core machine when no third-party domain was hosted.

VCPUs in sender domain	2	3	4	8	2	3	1	4
VCPUs in receiver domain	2	3	4	8	3	2	4	1
Success	0	92	100	2	67	5	7	0
Sync failure	100	8	0	0	33	95	93	100
Bit error	0	0	0	98	0	0	0	0

has a VCPU with a larger weight, the sender's CPU load has a small effect on execution times of processes running in the receiver domain. This results in failures in receiving communication requests. When the sender domain has a larger weight, the sender has too large an effect on execution times of processes running in the receiver's domain; specifically, the receiver process is likely to miss a communication request. In the current protocol, the receiver recognizes a communication request when the period of high CPU load lasts for about 1.5 s. However, a receiver running in a domain with an extremely small weight may consider that the high CPU load lasts for more than 1.5 seconds, and consequently, it may miss the communication request.

We report the result of preliminary experiments on a multi-core machine. The platform was a desktop personal computer with a Core 2 Quad 3 GHz and 4 GB main memory. We assigned various numbers of VCPUs to a sender domain and a receiver domain. Mapping between VCPUs and physical cores was not fixed, and hence each VCPU could run on any of the cores. The ratio between the sender and receiver weights was 1:1 in all cases. We attempted communication by CCCV 100 times.

The result is shown in Table 2. A 100% success rate was achieved when both the sender and receiver domain had four VCPUs. That was an ideal case because both domains were expected to put loads on all (four) physical cores and interfere with each other. CCCV achieved a 92% success rate when both the sender and receiver had three VCPUs. In that case, synchronization failure occurred only when all (three) receiver processes failed to recognize a communication request. When one or more receiver processes recognized a communication request, they successfully obtained correct bits based on a majority algorithm. When the sender and receiver had two VCPUs, all communication attempts failed. This result was natural because the domain scheduler was likely to assign two distinct physical cores to each domain. When the number of VCPUs in the sender and the receiver were both eight, most communication resulted in bit errors. We assume that the bit errors were caused by competition for physical cores among processes in a receiver group. When the sender and the receiver had a different number of VCPUs, success rates were low. We currently hypothesize that the most influential factor for high success rates is that the number of VCPUs in a *receiver* domain is close to the number of physical cores, and another influential factor is that the number of VCPUs in a *sender* domain is close to the number of physical cores.

Table 3: Accuracy of communication measured on a single-core machine when the Apache Web server was running in two third-party domains.

CPU load (%)	5-9	8-12	13-17
Success	92	91	74
Sync failure	7	9	13
Bit error	1	0	13

4.2.2 When Other Domains are Active

Next, we measured the accuracy obtained when the hypervisor hosted two third-party domains. In this experiment, the sender and receiver domains had the same weight. To raise the CPU load in the third-party domains, we ran the Apache Web server in both domains. We sent a number of requests to the Apache servers from another physical machine. All of the requests were for static Web pages.

We varied the CPU load of the third-party domains as follows:

CPU load 5-9: 55-60 requests per second were sent to one of the servers. The CPU load of the third-party domains and Dom0 varied between 5% and 9%.

CPU load 8-12: 55-60 requests per second were sent to both servers. The CPU load of the third-party domains and Dom0 varied between 8% and 12%.

CPU load 13-17: 55-60 requests per second were sent to one of the servers. 110-120 requests per second were sent to the other. The CPU load of the third-party domains and Dom0 varied between 13% and 17%.

Table 3 shows the results. Unlike the first experiment, we observed some errors in communicated bits. CCCV achieved

more than 90% accuracy when the CPU load of the third-party domains was less than 12%. However, when the average CPU load increased further, the accuracy dropped to 74%.

Synchronization failed in some attempts. A synchronization failure is likely to occur when both of the following conditions are satisfied:

- Third-party domains consume much CPU time when the receiver is executing an initialization procedure.
- Third-party domains consume little CPU time while the sender is sending a communication request.

Under these conditions, execution times of unitcomp in the synchronization phase can become shorter than the average execution time measured in an initialization procedure. As a result, the increase in execution times for synchronization is hidden by the increase in the initialization procedure.

4.2.3 Discussion

As described above, the receiver domain is more likely to fail in receiving a communication request when the weights of the sender receiver domains differ. Failure rates are even higher when the receiver's weight is larger. This problem is solved if the weight of the receiver domain can be set to a value not larger than the weight of the sender. As described in Section 3.1, we assume that a malicious program takes full control of the receiver domain. If the administrator of the virtual hosting service allows domain administrators to lower the weight of their domains, an attacker can do so without any difficulty. In general, no additional fee or negotiation is needed to lower the weight of a domain, because this benefits other domain users. A malicious program can obtain a low-weighted domain easily by choosing the most reasonable service package.

If the administrator of the hosting service forbids domain administrators from changing the weight of a domain and assigns different weights to different domains, CCCV must expect synchronization failures and bit errors. Even so, CCCV can achieve high accuracy if the sender has a larger weight than that of the receiver. The result of the first experiment showed that, even when the weight of the sender was three times larger than that of the receiver, communication had a success rate of over 90%. A moderate level of accuracy is obtained even when the sender has a smaller weight as long as the difference in weights is small.

The results in the second experiment showed that some synchronization failure and communication errors occurred when CPU usage occurred in other domains. However, when the CPU load was less than 12%, CCCV could achieve more than 90% accuracy. Success of communication depends on whether the CPU loads of processes other than the sender and receiver vary significantly during communication. Users of CCCV can minimize the effect of other processes by choosing an appropriate time during which the CPU load of other processes is low (e.g., late at night and early in the morning).

In the current protocol, the receiver judges the sender to raise the CPU load even when the receiver's performance is not much degraded. The threshold is 90% of the original performance. We chose the threshold to achieve successful communication even when the receiver has a larger weight (even when the receiver's execution times are unlikely to grow significantly). Meanwhile, this high threshold can cause a communication error due to a small performance degradation by execution of programs in third-party domains.

5. RELATED WORK

The work by Ristenpart et al. [11] shows an attack that identifies whether a particular virtual machine is likely to reside on the same physical server of a cloud infrastructure. It also demonstrates a side-channel attack that causes information leakage across virtual machines. The effectiveness of the attacks is evaluated on Amazon's EC2 platform. Although Ristenpart et al.'s work is the closest to our work, it differs in several points. First, we address a relaxed problem by assuming that an attacker can start a process in both communicating domains. We focus on designing a reliable communication protocol under that assumption. Second, our work adopts CPU loads as the communication medium, whereas their work adopts utilization of CPU caches.

The work by Wang et al. [18] shows a technique for creating a covert timing channel between virtual machines. The technique takes advantage of contention for an arithmetic logic unit (ALU) on simultaneous multithreading (SMT) processors. A sender using this technique transmits bits by controlling the time to use a shared ALU component (e.g., a multiplier). A receiver process recognizes the transmitted bits according to the time taken for using the component. Although the technique achieves a very high communication bandwidth, the target platforms are restricted to SMT processors.

Hu's papers [3, 4] describe other covert timing channels based on a delay due to bus contention or cache misses. They also describe several countermeasures to covert channel problems, such as reducing the precision of system clocks and adopting a secrecy-aware process scheduler. Since the target of the study is the VAX security kernel, a virtual machine monitor implemented on a traditional VAX machine, the effectiveness of the described technique on modern machines and virtual machine monitors is not clear.

Xen's characteristics in terms of performance isolation are evaluated and reported in a paper by Matthews et al. [8]. Their paper reports that the Xen hypervisor can effectively isolate the performance of multiple virtual machines. They report that even if one virtual machine generated an almost 100% CPU load, the performance of Web servers running on the other virtual machines was not affected by a large amount. Unlike their study, we focused on the performance characteristics obtained when multiple virtual machines attempt to consume the entire CPU time. In addition, we show a scheme for applying the characteristics to covert channel creation.

sHype [13] is a MAC-based security extension to Xen. sHype enables administrators to apply various security policies to virtual machines running on an sHype-extended hypervisor. One of the policies, the Chinese Wall policy, can prevent covert channel communication achieved by CCCV because the policy guarantees that a pair of specified virtual machines (i.e., domains) do not run at the same time on the same hypervisor. Unfortunately, administrators of sHype who apply the Chinese Wall policy to their domains must recognize and specify domains that may communicate via a covert channel. However, recognizing covertly communicating domains is not straightforward in the threat scenario described in Section 3.1. If the administrator successfully specified such domains, the Chinese Wall policy would bring

another disadvantage: it restricts domains that can run at the same time and results in loss of scalability.

Jaeger et al. [5] propose a policy model of the covert channel leakage problem in virtual-machine-based systems. Their model identifies information flows due to a combination of covert and overt channels. The policy for managing such flows is called a risk flow policy. They explore the application of various access control models, including the Chinese Wall model, to express risk flow policies. The focus of their study is to provide a concept for a risk flow policy and evaluate the ability of various access control models to express risk flow policies. A scheme for creating a covert channel is outside of their scope. Their research complements ours; it will greatly help to evaluate or formalize the risk of covert channels by CCCV.

6. SUMMARY AND FUTURE WORK

We developed and evaluated CCCV, a communication system across Xen virtual machine boundaries that uses CPU load as a covert channel. Since CCCV runs as user processes in guest OSes, the users do not need administrator privileges on a guest OS or the host OS. We confirmed experimentally that CCCV could communicate 64-bit data in 131.5 s with considerably high accuracy. CCCV could communicate data with a 100% success rate in an ideal environment where domains other than communicating domains were consuming little CPU time. Even when two third-party domains were hosting a Web server, CCCV achieved a success rate of over 90%.

Future research could follow several paths. The first is to conduct further experiments on other machines, other operating systems, and other virtualization infrastructure. For example, evaluating covert channels on container-based systems [6, 14] is an interesting future project. The second is to identify how much the bandwidth and accuracy of communication by CCCV can be improved. A key is to minimize or level off varying CPU usage by other processes. A simple improvement is to introduce redundancy in communicated information (e.g., to send the same 64-bit data 100 times with a 10-min interval or to adopt error correction code). The last one is to propose an effective scheme to prevent covert channel communication by CCCV or similar systems. A simple but promising scheme is to run a disturbing process that creates a random CPU load on the same hypervisor.

7. ACKNOWLEDGMENTS

This work was supported by KAKENHI 19700024.

8. REFERENCES

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 164–177, 2003.

[2] T. G. Handel and M. T. S. II. Hiding Data in the OSI Network Model. In *Proceedings of the 1st International Workshop on Information Hiding*, volume 1174 of *Lecture Notes in Computer Science*, pages 23–38, 1996.

[3] W.-M. Hu. Reducing Timing Channels with Fuzzy Time. In *Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*, pages 8–20, 1991.

[4] W.-M. Hu. Lattice Scheduling and Covert Channels. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy*, pages 52–61, 1992.

[5] T. Jaeger, R. Sailer, and Y. Sreenivasan. Managing the Risk of Covert Information Flows in Virtual Machine Systems. In *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies*, pages 81–90, 2007.

[6] M. Lageman. Solaris Containers — What They Are and How to Use Them. `http://www.sun.com/blueprints/0505/819-2679.pdf`, 2005.

[7] B. W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, 16(10):613–615, 1973.

[8] J. N. Matthews, W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, G. Hamilton, M. McCabe, and J. Owens. Quantifying the Performance Isolation Properties of Virtualization Systems. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, 2007.

[9] National Computer Security Center. A Guide to Understanding Covert Channel Analysis of Trusted Systems. Technical Report NCSC-TG-030, 1993.

[10] D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: The Case of AES. In *Proceedings of the Cryptographer's Track at the RSA Conference 2006*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20, 2006.

[11] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009)*, 2009.

[12] RSA Security Inc. RSA Data Loss Prevention (DLP) Suite. `http://www.rsa.com/node.aspx?id=3426`.

[13] R. Sailer, T. Jaeger, E. Valdez, R. Cáceres, R. Perez, S. Berger, J. L. Griffin, and L. van Doorn. Building a MAC-based Security Architecture for the Xen Opensource Hypervisor. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 276–285, 2005.

[14] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 275–287, 2007.

[15] Symantec Corporation. Symantec Data Loss Prevention. `http://www.symantec.com/business/data-loss-prevention`.

[16] TrendMicro. LeakProof. `http://us.trendmicro.com/us/products/enterprise/leakproof/`.

[17] VMware. VMware vShield Zones. `http://www.vmware.com/jp/products/vshield-zones/`.

[18] Z. Wang and R. Lee. Covert and Side Channels due to Processor Architecture. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 473–482, 2006.