Regular Paper

Checkpointing an Operating System Using a Parapass-through Hypervisor

Yoshihiro Oyama[1,a) Yudai Kawasaki[1,b) Kazushi Takahashi[1,c)

Received: June 14, 2014, Accepted: December 3, 2014

Abstract: Many dynamic malware analysis systems based on hypervisors have been proposed. Although they support malware analysis effectively, many of them have a shortcoming that permits the malware to easily recognize the virtualized hardware and change its execution to prevent analysis. We contend that this drawback can be mitigated using a hypervisor that virtualizes the minimum number of hardware accesses. This paper proposes a hypervisor-based mechanism that can function as a building block for dynamic malware analysis systems. The mechanism provides the facility for checkpointing and restoring a guest OS. It is designed for a parapass-through hypervisor, that is, a hypervisor that runs directly on the hardware and does not execute a host OS or an administrative guest OS. The advantage of using a parapass-through hypervisor is that it provides a virtual machine whose hardware configuration and behavior is similar to the underlying physical machine, and hence, it can be stealthier than other hypervisors. We extend the parapass-through hypervisor BitVisor with the proposed mechanism, and demonstrate that the resulting system can successfully checkpoint and restore the states of Linux and Windows OSes. We confirm that hypervisor detectors running on the system cannot identify the virtualized hardware, and determine that they are executing on a physical machine. We also confirm that the system imposes minimal overhead on the execution times of the benchmark programs.

Keywords: hypervisor, virtual machine monitor, checkpointing, malware

1. Introduction

Considerable literature has been published on dynamic malware analysis systems based on emulators and hypervisors [5], [11], [12], [19]. Although these systems effectively support malware analysis, many of them have a shortcoming that permits the malware to detect that it is running on an emulator or hypervisor. They can then behave differently or abort to prevent their behavior from being exposed to analysis [1], [24]. Methods of detecting emulators or hypervisors from inside a guest OS are well studied [7], [8], [23] and frequently adopted in malware. Therefore, malware analysis systems designed to execute malware must be stealthy. Providing a practically real environment to malware execution is an effective method to achieve stealthy analysis systems. This is also advantageous in the analysis of malware that does not attempt to detect an emulator or hypervisor because such malware could exhibit unintended behavior owing to the virtualization. Unexpected changes of this nature prevent comprehensive malware analysis. Chen et al. [4] reported that at least 4% of their malware samples demonstrated less malicious behavior under virtual machine executions. Lau et al. [16] reported that 2.13% of their malware samples were aware of virtual machines.

Parapass-through hypervisors are promising for stealthy malware analysis because they create a virtual machine whose specification and behavior is almost the same as that of the underlying physical machine and hence become stealthier than other popular hypervisors such as Xen and KVM. In the parapass-through architecture, the majority of the I/O operations pass through the hypervisor and only minimum operations are intercepted by the hypervisor. Most devices are not virtualized and are controlled by device drivers in the guest OS.

BitVisor [25] is the most popular parapass-through hypervisor. It is designed for security enhancement. The structure of BitVisor is illustrated in **Fig. 1**. BitVisor executes directly on the hardware and usually hosts only one guest OS at one time. It provides parapass-through device drivers that intercept I/O requests and responses to insert additional operations such as security enforcement. The BitVisor core and BitVisor extensions provide a wide range of security facilities such as VPN, background storage encryption [20], and malware signature detection [21]. The major advantage of BitVisor is a small TCB (Trusted Computing

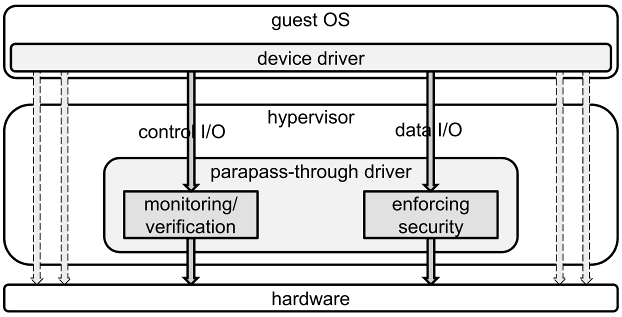

Fig. 1 Structure of BitVisor.

[1] The University of Electro-Communications, Chofu, Tokyo 182–8585, Japan

a) oyama@inf.uec.ac.jp

b) yudai@ol.inf.uec.ac.jp

c) kazushi@inf.uec.ac.jp

Base). The security facilities of BitVisor are not dependent on the guest OS and the BitVisor code is significantly smaller than the OS code. Shinagawa et al. [25] reported that BitVisor had only 21,582 lines of code.

Despite its potential in a stealthy infrastructure for malware analysis, BitVisor does not provide a snapshot or checkpointing facility. Consequently, BitVisor is not convenient for malware analysis because users have difficulty in restoring the execution environment to a normal state after malware execution. Unfortunately, no existing study has determined a method to extend a parapass-through hypervisor with a checkpointing facility.

In this paper, we propose a mechanism for parapass-through hypervisors that checkpoints and restores guest OS states. We implement BVCP, a BitVisor-based system into which the mechanism is incorporated, and demonstrate through experiments that BVCP functions well with widely used OSes and commodity computers. We also confirm that the mechanism imposes a small runtime overhead. Although this study focuses on parapass-through hypervisors, we expect that the insights and techniques of this paper are useful in extending other hypervisors with a checkpointing mechanism that depends minimally on a host OS or an administrative guest OS.

Several challenges must be addressed to integrate a checkpoint and restore mechanism in a parapass-through hypervisor. The mechanism must find a location for storing the checkpointed data. The method for a parapass-through hypervisor to save data to or load data from storage is not a clearly defined function because the hypervisor cannot get support of a host OS. The hypervisor has no file system where it can store the checkpointed data. This situation differs from most hypervisors that run a host or administrative guest OS. Moreover, the mechanism must interact with the physical hardware devices because the hypervisor executes directly on the hardware; it must include the low-level code to manipulate raw devices to transfer the checkpointed data to storage.

The contributions of this study are:

- It is the first study to propose a method of incorporating a checkpointing mechanism into a hypervisor without depending on a host or administrative guest OS.
- It demonstrates that the proposed mechanism can work effectively with widely used OSes and computers and that the mechanism can present actual hardware to programs running in a virtual machine.

One advantage of achieving the mechanism without depending on a host or administrative guest OS is that the resulting system can provide a more stealthy environment. If a host OS and/or another guest OS is running on the hypervisor, the resource view observed by malware is varied because of resource virtualization or resource consumption changes. For example, the kernel or system daemons in another guest OS consume CPU cycles and physical memory pages, and consequently malware may recognize that less resource is actually assigned to its execution environment. In addition, the introduction of another guest OS often results in the provision of virtual hardware different from the real one (e.g., virtual hard disk). Another advantage is that it improves user experiences in the primary usage assumed by BitVisor. BitVisor was originally intended to be used in daily computer uses as an almost transparent infrastructure for enforcing security facilities. The introduction of the checkpointing mechanism enables OS rollback without additional programs or kernel modification. OS rollback is useful to those other than malware analysts; usage examples include installing software that is not completely trusted, and running programs whose execution trails should be erased from the computer to keep privacy.

This paper is organized as follows. Section 2 presents a brief explanation of the major techniques for hypervisor detection. Section 3 describes the proposed system. Section 4 discusses our design decisions and several points requiring user consideration. Section 5 reports the experimental results. Section 6 describes related work and Section 7 provides a brief summary of this paper and directions for future research.

2. Hypervisor Detection

Many studies have been proposed concerning techniques for hypervisor detection [4], [6], [7], [8], [9], [10], [23], [28]. Malware can also utilize the techniques to hide from malware analyzers and detectors. Major detection techniques are as follows:

(1) **Detection of virtual hardware:** This technique searches for virtual hardware that indicates the existence of a hypervisor. Virtual hardware can be identified by obtaining the product information of the devices or executing I/O operations to manipulate the hardware devices. OS users can obtain the vendor names of the hardware devices by searching for a keyword from the system logs or executing OS management utilities. The MAC address assigned to a network card can also reveal that a network card is virtual [7].

(2) **Detection of I/O backdoor:** This technique issues a special I/O request and observes the result. Some hypervisors provide a set of I/O ports through which a program in a guest OS communicates with the hypervisor [3], [9]. The behavior of I/O port operations differs between a virtual machine and a real machine.

(3) **Detection of changes in instruction execution:** This technique examines the instruction execution of a virtual CPU and identifies a behavior different from a real CPU. Some hypervisors do not correctly emulate specific instructions for performance purposes [3], [7]. Bugs in a real CPU or in an emulator can also be used for detection [22], [23].

(4) **Detection of changes in timing:** This technique measures the execution times of special operations and determines that a hypervisor is operating underneath if the execution times are slower than those expected on a real machine.

The proposed study focuses on preventing hypervisor detection using the first technique, detection of virtual hardware, because it is simple and easy to implement in malware. Although the timing-based technique is more robust, we surmise that many malware developers will not choose this because it requires an additional development effort and can return an erroneous result owing to the variable load of other applications. We expect that the proposed system and other stealthy analysis systems supplement each other. We propose that malware analysts use the proposed system if they cannot analyze malware on other analysis systems (e.g., VMware-based) because of virtual hardware detection by the malware.

3. Proposed System

3.1 Overview

Users begin by booting BVCP (an extended BitVisor) and then boot a guest OS over it. When they choose BVCP in a boot loader such as GRUB, BVCP is started and it executes the boot loader a second time in a newly created virtual machine. The users then select a guest OS in the boot loader, and the guest OS boots. Upon completion of the boot process, the users are generally unable to notice that the guest OS is executing on a virtual machine. The appearance and behavior of the guest OS is the same as when it is running directly on the underlying physical machine. The OS kernel running on BVCP recognizes the same set of hardware devices and BIOS as the ones detected when running on the physical machine.

Figure 2 describes the approach employed by BVCP for malware analysis. To begin, a malware analyst executes a guest OS for malware execution on top of BVCP and then checkpoints the OS in its normal state. The analyst then executes malware and observes the result. The malware may destroy critical files, install malicious services, or inject malicious code into a running process. After examination, the analyst restores the guest OS. The guest OS state, which can be infected with malware or corrupted, is discarded. The guest OS restarts from the execution point at which the OS is checkpointed.

In a checkpointing operation, BVCP saves the guest OS state and continues the execution of the guest OS. Currently, BVCP can maintain only one checkpoint. One checkpoint can be restored multiple times; an analyst can examine an interested period of malware execution an arbitrary number of times. In the restoration operation, BVCP restores the checkpoint data and consequently, the guest OS rolls back to the checkpointed state. All modifications to memory or disk performed after the checkpointing are discarded.

3.2 Checkpointing Memory Data

BVCP saves and restores the data resident in CPU registers, memory, and on disks. Memory data and register values are saved to a free memory area that is allocated by the hypervisor at boot time. BVCP users specify the physical memory size of the guest OS to be less than half of the physical memory size of the real hardware. They also stipulate the start physical address of the free memory area. This must be greater than the middle point of the address range of the physical memory hardware.

BVCP checkpoints the data residing in the physical memory of the virtual machine. It saves the entire portion of the physical memory except for specific ranges. One of the bypassed ranges is the code and data area of the hypervisor. We use a function in the original BitVisor code that returns the start and end address of this range. A second omitted range is reserved by BIOS and thus, unavailable to a guest OS. BVCP determines the reserved and unreserved memory ranges from a physical memory map obtained by issuing BIOS-e820 at boot time. **Figure 3** lists an example of boot-time messages for the physical memory map. The memory ranges labelled "`usable`" are saved; the others are not.

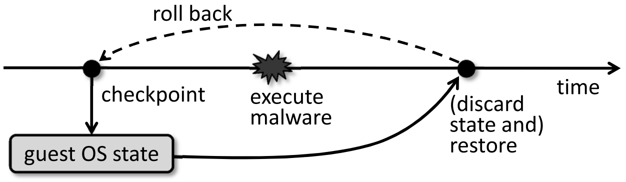

Fig. 2 Using BVCP for malware analysis.

Copying the content of a given physical memory range to free memory is not straightforward because the hypervisor must access memory with virtual addresses. Hence, the hypervisor manipulates the page table to create virtual pages that are mapped to the saved physical pages. It then copies the content of the memory range to free memory using the `memcpy` function with the newly obtained virtual address as an argument. Details are described in Section 3.4.

BVCP also saves the CPU states of the virtual machine. In particular, it saves (1) the values in the guest-state area of a VMCS structure in Intel VT-x, and (2) a register value not included in the structure (i.e., a value of the RBP register). BVCP saves all the fields in the guest-state area, which contains tens of states including the CPU register values, a physical address range, interruptibility states, and activity states. The saved CPU registers include most special-purpose registers such as RSP, RIP, RFLAGS, CRx, segment selector registers, and table pointers. However, the saved CPU registers do not include the base pointer register RBP. Therefore, we added code for saving the RBP value.

BVCP restores the previous guest OS state by loading the checkpointed data to the appropriate memory areas and registers. The physical memory of the virtual machine is simply overwritten by the checkpointed data. Several of the entries of the VMCS structure of the virtual machine are reloaded with the saved values using the `VMWRITE` instruction.

3.3 Checkpointing Disk Data

To checkpoint the disk data, BVCP users create at least two partitions on a disk in advance. One partition (*main partition*) is the normal storage for a guest OS. The second partition (*diff partition*) stores the differential data written by the guest OS after checkpointing. BVCP switches the destination of the disk accesses between the main partition and the diff partition (**Fig. 4**). The switching operations are executed by a software module called *access switch*. Access switch intercepts disk I/O and switches the destination of the disk accesses according to the accessed disk blocks. Before checkpointing, requests for both

```
00000000-0009d7ff, usable     da69b000-da6ddfff, ACPI NVS
0009d800-0009ffff, reserved   da6de000-dadcefff, usable
000e0000-000fffff, reserved   dadcf000-dafdcfff, reserved
00100000-1fffffff, usable     dafdd000-daffffff, usable
20000000-201fffff, reserved   db800000-df9fffff, reserved
20200000-3fffffff, usable     f8000000-fbffffff, reserved
40000000-401fffff, reserved   fec00000-fec00fff, reserved
40200000-d9cf7fff, usable     fed00000-fed03fff, reserved
d9cf8000-da415fff, reserved   fed1c000-fed1ffff, reserved
da416000-da695fff, ACPI NVS   fee00000-fee00fff, reserved
da696000-da69afff, ACPI data  ff000000-ffffffff, reserved
```

Fig. 3 Example of physical memory maps obtained by BIOS-e820.

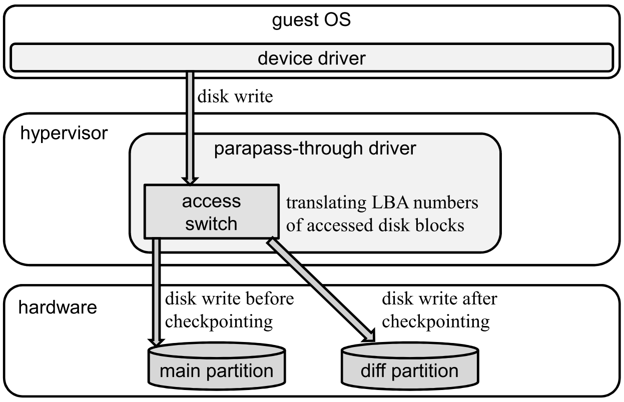

Fig. 4 Switching disk accesses between different partitions for future restoration of the checkpointed OS state.

disk reads and disk writes pass through the access switch and are sent to the main partition; the diff partition is not used. Once the guest OS is checkpointed, the diff partition is activated and thereafter, the access switch forwards all disk-write requests to the diff partition. The destination of the disk-read requests is switched between the two partitions based on whether the disk block to be read is clean (not modified after checkpointing) or dirty (modified after checkpointing).

The mechanism assumes that the guest OS addresses disk blocks with LBA (Logical Block Addressing) numbers. The access switch modifies the LBA number contained in a disk I/O request and thus changes the accessed disk block. Given an LBA number in the main partition and an operation type (i.e., read or write), the access switch determines the appropriate partition and calculates the corresponding LBA number in the partition. It maintains a *dirty block table* to manage the LBA numbers of dirty blocks.

The guest OS may wish to read a sequence of disk blocks that contain both clean and dirty blocks. Therefore, when the access switch intercepts a request for reading disk blocks, it first creates and issues a new request to read the clean blocks from the main partition. The access switch also creates and issues a new request to read the dirty blocks from the diff partition. After collection of the blocks from both partitions, it returns a notification of disk-read completion to the guest OS.

When a restoration of the guest OS is initiated, the access switch resets the destination of all disk operations to the main partition. The data in the diff partition is discarded.

In the following, we explain the translation of LBA numbers with an example. We suppose that the main partition begins with the LBA number `0x100000` and the diff partition begins with the LBA number `0x6500000`. The distance between the corresponding blocks in these partitions is `0x6400000`. Then, we assume that a guest OS sends an I/O request for writing data in a sequence of disk blocks whose LBA numbers are from `0x130000` to `0x13ffff`. The access switch translates the LBA numbers `0x13****` into `0x653****` and raises dirty flags in the corresponding entries in the dirty block table. Now, we assume that the guest OS sends an I/O request for reading data from a sequence of disk blocks whose LBA numbers are from `0x120000` to `0x14ffff`. The access switch translates the LBA numbers and consequently, the guest OS actually reads the data from LBA numbers `0x6520000` to `0x654ffff`. When the read I/O completes, a notification of disk read is sent to the guest OS. The access switch intercepts the notification, creates a new I/O request for reading clean blocks based on the dirty block table, and sends the request to the disk. In this case, it reads two sequences of disk blocks whose LBA numbers range from `0x120000` to `0x12ffff`, and from `0x140000` to `0x14ffff`. The appropriate portion of the first read data are overwritten with the newly read data. Finally, the access switch passes the intercepted notification to the guest OS.

```
void map_and_memcpy(u64 src_p, u64 len)
{
  /* virtual addresses of source and destination */
  u8 *src_v, *dst_v;
  /* pre-allocated area for storing checkpointed data */
  u8 *save_area;

  save_area = ...;
  src_v = mapmem_hphys(src_p, len, 0);
  dst_v = mapmem_hphys(save_area + src_p, len, 0);

  memcpy(dst_v, src_v, len);

  unmapmem(src_v, len);
  unmapmem(dst_v, len);
}

void checkpoint_memory_data(u64 area, u64 len)
{
  u64 limit = area + len;
  u64 b = area;
  while (b < limit) {
    if (out_of_hypervisor_range(b)) {
      u64 copied_len = ...; /* block size, basically */
      map_and_memcpy(b, copied_len);
    }
    b += block_size;
  }
}
```

Fig. 5 Extract of a portion of the modified code for checkpointing memory data.

3.4 Implementation Detail

We first explain the detail of checkpointing memory data. BVCP saves the content of memory areas which are indicated as "`usable`" in a physical memory map and are within the address range assigned to the guest OS. **Figure 5** shows the code added for checkpointing memory data.

BVCP copies the content to a pre-allocated memory area (`save_area` in the figure) at the granularity of *block*. The block size is 64 KB in the current implementation. BVCP checks whether each copied block is in the hypervisor area, and copies blocks that are out of the hypervisor area only. For each copied block, BVCP modifies the hypervisor's page table to create (1) contiguous virtual pages that are mapped to the physical pages in the copied block and (2) contiguous virtual pages that are mapped to the physical pages in the pre-allocated memory area. Then it copies the content from the former virtual pages to the latter virtual pages with ordinary memory transfer instructions in `memcpy`. After that, it modifies the page table again to delete the mapping for the virtual pages. BVCP uses `mapmem_hphys` to create virtual pages, and uses `unmapmem` to delete mapping. Both mapmem_hphys and unmapmem are functions originally provided by BitVisor.

When restoring the guest OS, BVCP performs an symmetric operation, which copies the content from the pre-allocated area to the memory area for the guest OS. The method of page mapping and memory copying used in the restoration operation is the same as the method used in the checkpointing operation.

We then explain the detail of checkpointing disk data. We modified the BitVisor code that manages storage I/O, specifically, I/O of ATA (Advanced Technology Attachment) and AHCI (Advanced Host Controller Interface). We modified the functions that process the DMA read or DMA write commands of ATA. The commands adapted were READ DMA, READ DMA EXT, WRITE DMA, and WRITE DMA EXT. Moreover, we modified the storage accesses using NCQ (Native Command Queuing). **Figure 6** is an extract of a portion of the modified code. The function ahci_handle_cmd_rw_dma is one of the aforementioned functions. Based on the provided arguments, it calculates the LBA number and number of sectors sent to storage. We added the function call switch_access_dst(port, ..., rw); to the function body. The function switch_access_dst was included to implement the access switch.

```
void ahci_handle_cmd_rw_dma(..., struct ahci_port *port,
                              ..., int rw, ...)
{
  u64 lba; /* LBA number */
  u32 nsec; /* number of sectors */
  lba = ...;
  nsec = ...;
  ...
  switch_access_dst(port, ..., rw);
}

void switch_access_dst(struct ahci_port *port, ...,
                       int rw)
{
  ...
  if (is_switching_mode()) {
    /* we are between checkpoint and restart */
    u64 org_lba = port->my[...].dmabuf_lba;
    u64 count = port->my[...].dmabuf_nsec;
    new_lba = !rw ? translate_lba_read(org_lba, count)
                  : translate_lba_write(org_lba, count);
    port->my[...].dmabuf_lba = new_lba;
    ...
  }
}

u64 translate_lba_read(u64 org_lba, u64 count)
{
  if (out_of_main_partition(org_lba)) {
    return org_lba;
  }
  if (!contain_dirty_page(org_lba, count)) {
    return org_lba;
  }
  return org_lba + partition_distance;
}

u64 translate_lba_write(u64 org_lba, u64 count)
{
  if (out_of_main_partition(org_lba)) {
    return org_lba;
  }
  set_dirty_flags(org_lba, count);
  return org_lba + partition_distance;
}
```

Fig. 6 Extract of a portion of the modified code for storage accesses.

The function switch_access_dst rewrites the LBA number in a given structure of type struct ahci_port. The function performs the rewrite if the current execution is between checkpointing and restoration. It uses different rewrite algorithms (translate_lba_read or translate_lba_write) based on whether the access is a read access or a write access. The function translate_lba_read determines if the accessed block contains a dirty page. If it does, it returns the original LBA number; otherwise, it returns a translated LBA number from the diff partition. The variable partition_distance, initialized when booting BitVisor, stores the distance between the main partition and the diff partition. The function translate_lba_write always returns a translated LBA number from the diff partition. It raises dirty flags for all the accessed blocks.

The users trigger checkpointing and restoration of the guest OS by executing a special program on the guest OS or pressing a special key. The hypervisor detects both the execution of the program and the press of the key combination, and reacts accordingly. The special program is a small function that merely invokes a hypercall to the hypervisor. We implemented two hypercalls, one for checkpointing and another for restoration. BVCP registers the handler function for the hypercalls at boot time. The program executes a VMCALL instruction after setting the hypercall ID to the RAX (or EAX) register. The key for checkpointing is assigned to F2 and the key for restoration is assigned to F4, by default. The original BitVisor implementation provides the code for intercepting keyboard interrupts (i.e., I/O port 0x60). Hence, we implemented the interception of F2 and F4 keys by extending this code. Users can change the key binding by changing the key code in the BVCP source code.

3.5 Difference from Other Hypervisors

The implementation of checkpointing in BVCP differs from the implementation in other hypervisors that have a host OS or administrative guest OS such as Xen and KVM.

First, a parapass-through hypervisor must allocate and manage memory and disk spaces to save checkpointed data because it originally has no space to save. Furthermore, because no other system software is running to manage the allocated spaces, the hypervisor must manage the space by itself. If a host OS is running, a hypervisor can save checkpointed data as a file on the host OS. However, in our setting, no file system is available and hence BVCP writes checkpointed data by manipulating the low-level requests issued to a hard disk. We decide to prepare a free disk area as a diff partition and use a latter half of physical memory to store checkpointed memory data. The design and implementation of the part is an originality of this work.

Second, a parapass-through hypervisor must read and write the state of real devices. Other hypervisors including Xen and KVM provide virtual hardware to a guest OS, and the hypervisors can completely control the state of the virtual hardware. On the other hand, in the case of a parapass-through hypervisor, no device abstraction is provided between a guest OS and real hardware, and the control of real hardware is not as simple as virtual hardware. Handling real devices is a large challenge to achieve checkpointing in a parapass-through hypervisor, and is discussed further in Section 4.

Third, a parapass-through hypervisor does not provide users with an execution environment for controlling a guest OS. Users of KVM or Xen can control a guest OS from an administrative software running in a host OS or an administrative guest OS on the hypervisor. Devising a method of triggering checkpointing and restoration is a challenge. BVCP expects users to trigger them by issuing a hypercall or pressing a special key. Implementing the triggering part and demonstrating that it works with no problem is also an originality of this work.

4. Discussion

4.1 Device States

To capture complete guest OS states, the internal states of the hardware devices must be extracted and saved. However, some of these internal states are stored in write-only device registers or obtained only by issuing I/O operations to the device. They accompany an execution of an `in` and/or `out` instruction on Intel x86 architectures. A network card, hard disk, and graphics card have their own states that are not included in the main memory or CPU registers.

The current version of BVCP does not save or load the internal state of the hardware devices. Physical memory ranges for memory-mapped I/O (MMIO) are indicated as reserved and memory accesses to these ranges are transformed to I/O accesses. In actuality, data in these ranges are not memory data and cannot necessarily be saved or loaded with ordinary memory access instructions. Hence, BVCP does not save or load them. Consequently, hardware states at the time of restoration can be inconsistent compared to the checkpointed data. Nevertheless, in many of our experiments, a guest OS continued to function successfully after restoration. Although a desktop screen can become distorted after restoration owing to the inconsistency between the graphics card and the guest OS, it recovers a normal appearance when the guest OS next repaints the screen. Based on our experience, the state of some devices such as graphics cards does not require saving and restoring.

Undoubtedly, there is a possibility that an inconsistency could cause a problem or fault for the guest OS. In the worst case, the guest OS could terminate abnormally. In some cases in our experiments, the network did not function after restoration. BVCP requires further development to achieve more reliability in terms of the restoration of the device states. In the current implementation, the restoration of the disk data is more reliable than the restoration of the memory data. Hence, BVCP provides a method to allow users to deactivate memory data checkpointing and use disk data checkpointing only.

4.2 Number of Checkpoints

The current version of BVCP can retain only one checkpoint at any one time. We believe that the support of one checkpoint satisfies the minimum requirement for the infrastructure of malware analysis. We avoid supporting multiple checkpoints because the management of multiple checkpoints would require considerable extension to the BVCP implementation and increase the runtime overhead. To support multiple checkpoints, the BVCP implementation would need to manage multiple diff partitions or create multiple "logical" diff partitions in one "physical" diff partition. Choosing an appropriate diff partition from multiple diff partitions and calculating the correct LBA number would be a complicated operation. However, the support of multiple checkpoints could be useful in some malware analysis and we identify this support as a future work.

4.3 Network

BVCP does not intercept or virtualize network communication. If malware running on a guest OS rolls back, an external server communicating with the malware does not roll back. Hence BVCP users must be aware that malware can behave differently after the restoration of the guest OS. Users should consider combining BVCP with other security systems such as sandboxes or packet filters to prevent malware execution from damaging external machines.

4.4 Physical Memory View

Some sophisticated malware may attempt to detect a hypervisor by checking the physical memory size or physical memory map assigned to a virtual machine. If the size or map is different from typical ones observed in real environments, the malware has a reason to increase the probability of being running on a virtual machine. BitVisor and BVCP provide a virtual machine with a spurious physical memory map different from a real one because a hypervisor resides in some parts of physical memory. The physical memory size available to a virtual machine also differs from the size available to a real machine.

Currently, BVCP does not provide a countermeasure against this type of hypervisor detection. However, we expect that an extension to BVCP will harden the detection significantly. The extension hides a hypervisor by moving its memory parts to the outside of the spurious physical memory range, and further changes the physical memory size and physical memory map to plausible ones.

5. Experiments

5.1 Test of Checkpointing

We implemented BVCP based on BitVisor 1.3 and tested its behavior with experimentation. The hardware platform used in all experiments was the desktop PC described in **Table 1**. We assigned one CPU core and 4 GB memory to a virtual machine. The guest OSes used in the experiments were Ubuntu 12.04 (32 bit), Fedora 20 (32 bit), and Windows 7 (32 bit).

We executed a terminal and web browser on a guest OS and checkpointed the OS. We then terminated the terminal and browser and restored the OS. The states of the guest OS and applications, including the displayed contents, locations of win-

Table 1 Platform for experiments.

CPU	Intel Core i3-2120 3.3 GHz
Memory	8 GB
HDD	Seagate ST500DM002-1BD14 SATA 500 GB, 7,200 rpm
Chipset	Intel 7 Series/C210 Series Chipset

dows, and browsing history, were successfully rolled back to the previous states on all three OSes. There were no problems encountered with the use of the restored OS. On Windows 7, the desktop screen was not automatically repainted after restoration. However, when the user moved the mouse cursor, the screen areas around the mouse cursor were repainted.

In the second experiment, we checkpointed a guest OS and then deleted all the files placed on the desktop. When we restored the OS, all the files reappeared on the desktop, all with the correct content and file name.

Finally, we checkpointed a guest OS and then installed the Opera web browser. After the installation, we restored the guest OS. We confirmed that after the restoration, Opera did not appear in the list of installed applications. These experiments were successful on all of the OSes and there were no problems after the restoration.

We also confirmed that BVCP successfully checkpointed and restored guest OS states on a notebook PC, a TOSHIBA dynabook R631.

5.2 Hypervisor Detection

We tested the stealthy characteristic of BVCP using two tools and one algorithm for detecting hypervisors. These are publicly available from the web. The first tool is slabbed-or-not [27]. It combines several techniques for detecting hypervisors, such as checking I/O backdoors, CPU information, device information, and special files under `/proc`.

The second tool is virt-what [30], distributed through the Red Hat website. The tool prints a list of facts regarding a virtual machine running outside the OS that it derives from heuristics. It examines similar information sources to slabbed-or-not. For example, it checks CPUID, files under `/proc`, and the results from `dmidecode` and `uname`.

We also used the algorithm adopted in virtdetect [26]. This is a command for detecting if the OS is running on a virtual machine. The command is based on `Sys::Detect::Virtualization`, a widely distributed Perl package for hypervisor detection. The algorithm adopted in the command and the package determines the existence of a hypervisor by detecting virtual devices using three methods. The first reads the content of the kernel buffer with the `dmesg` command and searches for the keyword "`virtual`." The second reads the BIOS information with the `dmidecode` command and searches for character strings that represent product names of virtual hardware or hypervisor manufacturers. The final method reads the virtual files under `/proc` to obtain the name and specification of the IDE and SCSI devices.

We attempted to detect BVCP using the tools and algorithm from within a guest Ubuntu 12.04 OS. Slabbed-or-not could not detect a hypervisor. It displayed the following messages:

```
Not running under any known container type
Not running under any known hypervisor type
```

Virt-what could not detect a hypervisor. It did not display a message, indicating that it could not identify any evidence of a hypervisor. The virtdetect algorithm was unable to detect a hypervisor. The result of `dmesg` and `dmidecode` contained the information of the real hardware only, and did not contain a string indicating virtual hardware. The file `/proc/scsi/scsi` contained the name of the real HDD and DVD drives with no signs of virtualization.

Table 2 Time taken for checkpointing and restoration.

	checkpoint	restore
Ubuntu 12.04	492 ms	492 ms
Windows 7	493 ms	493 ms

5.3 Time for Checkpointing and Restoration

We measured the time taken for checkpointing and restoring a guest OS. The guest OS used in this experiment was Ubuntu 12.04 and Windows 7. We inserted to BVCP a code fragment for obtaining the current time before and after the function calls for checkpointing and restoration. To obtain the current time, we used the function `get_time()`, which was originally provided by BitVisor. Checkpointing and restoration perform an extremely small operation in terms of disk data; they just modify the variable that specifies the destination of disk accesses. Therefore, the time reported here is close to the time for checkpointing and restoring memory data.

Table 2 shows the result. The times taken for checkpointing or restoration were less than one second. The amount of memory data copied from the guest OS was approximately 2.46 GB in both of the Ubuntu and Windows cases. We believe that checkpointing and restoration are completed sufficiently fast, and hence BVCP users will rarely feel stress to their overhead.

Almost the same amounts of time were taken in the cases of Linux and Windows. It is natural because the cost of checkpointing and restoration by BVCP is dependent on the physical memory size of a checkpointed guest OS and is little affected by guest OS types or applications running in a guest OS. It is also natural that almost the same amounts of time were taken for checkpointing and restoration because they are symmetric operations with similar costs.

5.4 Benchmark Results

5.4.1 Setting

We measured the runtime overheads imposed by BVCP to evaluate its practicality. The guest OS used in this experiment was Ubuntu 12.04. The following execution environments were compared:

Native: An OS runs on a physical machine.

BitVisor: An OS runs as the guest OS over the original version of BitVisor.

BVCP: An OS runs as the guest OS over BVCP.

The performance of BVCP was measured in the period between checkpointing and restoration. When we built the BitVisor and BVCP systems, we included compilation options to enable the default mechanisms of BitVisor. We disabled the storage encryption and VPN.

We measured the performance of the general operations using UnixBench 5.1.3, the I/O performance with the Bonnie++ 1.97.1, and the performance of a real-world application with a system build benchmark.

5.4.2 UnixBench

Figure 7 presents the result of UnixBench. The shown values

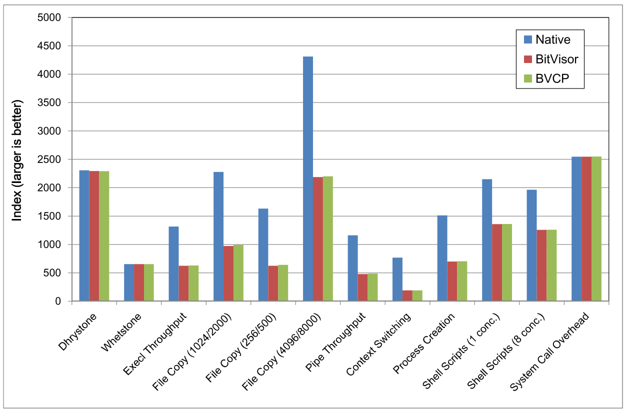

Fig. 7 Result of UnixBench.

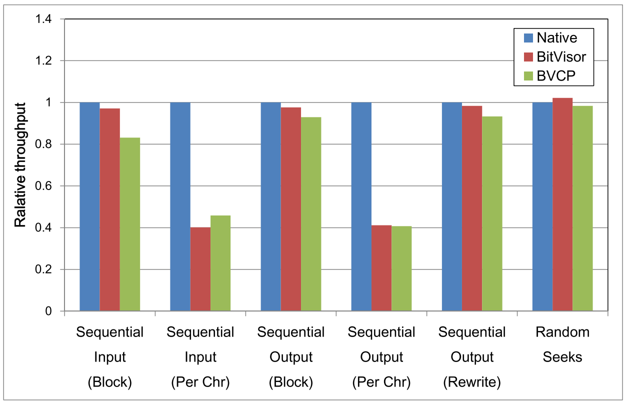

Fig. 8 Result of Bonnie++.

are the average values obtained in three executions. We discovered a significant difference in the indexes between Native and BitVisor, and a smaller difference between BitVisor and BVCP. A significant part of the runtime overhead of BVCP is probably caused by operations originally in BitVisor. In some sub-benchmarks, the performance of BVCP was slightly better than that of BitVisor. The performance of the benchmark fluctuated in different executions; BitVisor was better in some executions and BVCP was faster in other executions. Therefore, we surmise that better performance of BVCP in the result might be caused by the fluctuation. Another possible reason is changes of hard disk scheduling caused by changes of accessed sectors and I/O request timing.

5.4.3 Bonnie++

Figure 8 shows the result of Bonnie++. The shown values are the average values obtained in three executions. Each group in the figure indicates the amount of time taken for the following operations:

Random Seeks: Random accesses to the disk with the `lseek`, `read`, and `write` system calls.

Sequential Input Block: Sequential read of data blocks with the `read` system call.

Sequential Input Per Chr: Per-character sequential read with the `getc` macro.

Sequential Output Rewrite: Rewriting each part of a file with the `read`, `lseek`, and `write` system call.

Sequential Output Block: Sequential write of data blocks with the `write` system call.

Sequential Output Per Chr: Per-character sequential write with the `putc` macro.

Table 3 Result of system build benchmark.

	Native	BitVisor	BVCP
sequential compilation	44.60 s	51.14 s	50.49 s
concurrent compilation	44.75 s	50.69 s	50.62 s

The values in the figure represent relative throughputs (the throughputs of Native are 100%). Larger values indicate better performance.

In the execution of BitVisor and BVCP, per-character disk accesses imposed increased runtime overheads than per-block disk accesses. We believe that the reason is that the smaller access unit increases the number of disk I/O's and interceptions by the hypervisor. The throughputs of block operations of BitVisor were slightly lower than those of Native, and the throughputs of BVCP were still lower. The maximum performance degradation of block operations caused by BitVisor and BVCP were approximately 3% and 17%, respectively.

The throughput of BVCP was better than the throughput of BitVisor in Sequential Input Per Chr, and the throughput of BitVisor was better than the throughput of Native in Random Seeks. As in the UnixBench case, the performance fluctuated significantly in the executions of Bonnie++. In the two sub-benchmarks above, the execution environment achieving the best throughput varied in different executions.

Note that the benchmark performs I/O operations only and the experimental result indicates the worst-case overhead. BVCP imposed a 59% overhead on the Sequential Output Per Chr and it was the largest overhead among all sub-benchmarks. We emphasize that BVCP maintains the overhead to only 59% in the worst case.

5.4.4 System Build Benchmark

Table 3 indicates the time consumed for building BitVisor 1.3 with gcc 4.6.3. We attempted both a sequential and concurrent compilation. We executed the build benchmark after dropping all page cache by writing 3 in a special file `/proc/sys/vm/drop_caches`. The overheads of both BitVisor and BVCP were small. The time for BVCP was only 13.2% and 13.1% greater than the time of Native in sequential and parallel compilation, respectively. Based on these results, we expect that BVCP users will rarely recognize runtime overhead, and that malware will not detect hypervisors from the application performance information.

6. Related Work

Many general-purpose hypervisors support snapshots, including VMware Workstation, VirtualBox, KVM, Xen, and Hyper-V. These hypervisors virtualize more hardware devices than BitVisor, and hence they provide execution environments that can be more easily recognized by malware as virtual machines.

BareBox [15] is a Xen-based malware analysis system that can restore the saved states of a guest OS. Because Xen always executes a special virtual machine for resource management, BareBox takes advantage of the virtual machine to save checkpointed data. Ether [5] is also a Xen-based hypervisor for malware analysis. It cooperates with an administrative guest OS and monitors events occurring in a target guest OS. Because BareBox and Ether are based on Xen, more virtual devices are exposed to malware than in the case of BVCP. Further, Ether does not have a checkpointing mechanism. MAVMM [19] is a lightweight hypervisor that virtualizes a minimum set of hardware devices. It provides an execution environment that is similar to a physical machine. MAVMM provides several mechanisms for monitoring and recording the execution of a guest OS. MAVMM does not have a checkpointing mechanism.

Kang et al. [13] proposed a QEMU-based malware analysis system that dynamically modifies the execution of an emulator and deceives anti-emulation checks by malware. It adopts a sophisticated technique for diagnosing anti-emulation behavior based on a comparison of malware execution between an emulated platform and a reference platform. Although their system provides effective resistance against timing and CPU semantics attacks, they admit that their system would not be useful against attacks that use hardware characteristics. Conversely, BVCP targets malware that attempts hardware characteristic attacks. Malware analysts can improve their quality of analysis by combining their system and BVCP.

REFORM [28] is a malware analysis tool implemented as a C++ plugin to a widely used debugger IDA Pro. It scans the memory of a malware program to find code signatures for determining if VMware is running (they call these anti-VMware signatures). The tool automatically patches the execution results of the code to disguise that the malware is running in a VMware virtual machine. The tool adds a stealth characteristic to the debugger against malware that attempts to detect I/O backdoors and the anomalous behavior of instructions for accessing descriptor tables. However, unlike BVCP, it does not provide any countermeasure against malware that examines the name or behavior of hardware devices.

API Chaser [14] is a malware analysis system that provides anti-analysis resistant API monitoring. This system incorporates numerous techniques to prevent malware from evading analysis. It is based on QEMU and has artifacts for hiding the fact that the malware is running on a QEMU virtual machine. The artifacts change the product names of the virtual hardware and dynamically patch specific instruction patterns in the malware code for detecting QEMU. Although the artifacts change the product names of the virtual hardware, they do not change the behavior of the virtual hardware. Hence, virtualization by API Chaser is likely to be detected by malware that examines the behavior of hardware devices.

Running a hypervisor directly on hardware and permitting some I/O operations to pass through the hypervisor is a powerful architecture that provides the advantage of a low overhead and/or small TCB. A work by Liu et al. [18] applied the architecture to fast I/O operations of InfiniBand interconnects. The proposed study applies the architecture to stealthy malware analysis.

Alcatraz [17] creates an isolated execution environment for untrusted software. It isolates a file system by intercepting file operations issued by applications and changing given file paths. Modification to files is accumulated to a file tree different from the base file tree, and is simply discarded when a user restores the file system. Although Alcatraz runs inside an OS and modifies file operations, BVCP changes the accessed disk blocks from outside the OS, thus enabling checkpointing of all the OS states.

The ZFS [29] and Btrfs [2] file systems have a snapshot mechanism based on the copy-on-write method. Because they are file systems, they must replace the currently used file system. Conversely, BVCP is a hypervisor and can coexist with any file system and OS.

7. Summary and Future Work

This paper proposed a mechanism for a parapass-through hypervisor that enables checkpointing and restoring guest OS states. We incorporated the mechanism into BitVisor and conducted experiments using the resulting BVCP system. This introduced a 13.2% runtime overhead (maximum) to the system build benchmark.

Future research follows several paths. The reliability of BVCP could be further improved by addressing a problem concerning the internal state of the hardware devices. We are investigating a method of resetting the hardware devices to a regular state immediately prior to checkpointing. Furthermore, it would be advantageous to collect malware analysis information using BVCP. Examining the behavior of hypervisor-aware malware is particularly important. We can even create and examine artificial malware that attempts to recognize BVCP. In the long run, a scheme for quantitatively evaluating the stealthiness of hypervisor-based systems will be needed. Finally, an implementation scheme that extends BVCP to support multiple checkpoints should be considered.

Acknowledgments This work is supported by JSPS KAKENHI Grant Numbers 23700032, 26330080. Takaaki Fukai of University of Tsukuba provided helpful comments for our work.

References

[1] WORM_AGOBOT.GEN, available from ⟨http://about-threats.trendmicro.com/us/malware/worm_agobot.gen⟩.

[2] Btrfs, available from ⟨https://btrfs.wiki.kernel.org/⟩.

[3] Carpenter, M., Liston, T. and Skoudis, E.: Hiding Virtualization from Attackers and Malware, *IEEE Security & Privacy*, Vol.5, No.3, pp.62–65 (2007).

[4] Chen, X., Andersen, J., Mao, Z.M., Bailey, M. and Nazario, J.: Towards an Understanding of Anti-virtualization and Anti-debugging Behavior in Modern Malware, *Proc. 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp.177–186 (2008).

[5] Dinaburg, A., Royal, P., Sharif, M. and Lee, W.: Ether: Malware Analysis via Hardware Virtualization Extensions, *Proc. 15th ACM Conference on Computer and Communications Security*, pp.51–62 (2008).

[6] Egele, M., Scholte, T., Kirda, E. and Kruegel, C.: A Survey on Automated Dynamic Malware-analysis Techniques and Tools, *ACM Computing Surveys*, Vol.44, No.2 (2012).

[7] Ferrie, P.: Attacks on More Virtual Machine Emulators, Technical report, Symantec Advanced Threat Research (2007).

[8] Garfinkel, T., Adams, K., Warfield, A. and Franklin, J.: Compatibility is Not Transparency: VMM Detection Myths and Realities, *Proc. 11th Workshop on Hot Topics in Operating Systems* (2007).

[9] Holz, T. and Raynal, F.: Detecting Honeypots and other suspicious environments, *Proc. 2005 IEEE Workshop on Information Assurance and Security* (2005).

[10] Issa, A.: Anti-virtual machines and emulations, *Journal in Computer Virology*, Vol.8, No.4, pp.141–149 (2012).

[11] Jiang, X., Wang, X. and Xu, D.: Stealthy Malware Detection and Monitoring through VMM-Based “Out-of-the-Box” Semantic View Reconstruction, *ACM Trans. Information and System Security*, Vol.13,

No.2 (2010).

[12] Kang, M.G., Poosankam, P. and Yin, H.: Renovo: A Hidden Code Extractor for Packed Executables, *Proc. 5th ACM Workshop on Recurring Malcode* (2007).

[13] Kang, M.G., Yin, H., Hanna, S., McCamant, S. and Song, D.: Emulating Emulation-Resistant Malware, *Proc. 2nd ACM Workshop on Virtual Machine Security*, pp.11–22 (2009).

[14] Kawakoya, Y., Iwamura, M., Shioji, E. and Hariu, T.: API Chaser: Anti-analysis Resistant Malware Analyzer, *Proc. 16th International Symposium on Research in Attacks, Intrusions and Defenses*, Lecture Notes in Computer Science, Vol.8145, pp.123–143 (2013).

[15] Kirat, D., Vigna, G. and Kruegel, C.: BareBox: Efficient Malware Analysis on Bare-Metal, *Proc. 27th Annual Computer Security Applications Conference*, pp.403–412 (2011).

[16] Lau, B. and Svajcer, V.: Measuring virtual machine detection in malware using DSD tracer, *Journal in Computer Virology*, Vol.6, No.3, pp.181–195 (2010).

[17] Liang, Z., Sun, W., Venkatakrishnan, V.N. and Sekar, R.: Alcatraz: An Isolated Environment for Experimenting with Untrusted Software, *ACM Trans. Information and System Security*, Vol.12, No.3 (2009).

[18] Liu, J., Huang, W., Abali, B. and Panda, D.K.: High Performance VMM-Bypass I/O in Virtual Machines, *Proc. 2006 USENIX Annual Technical Conference*, pp.29–42 (2006).

[19] Nguyen, A.M., Schear, N., Jung, H., Godiyal, A., King, S.T. and Nguyen, H.D.: MAVMM: Lightweight and Purpose Built VMM for Malware Analysis, *Proc. 2009 Annual Computer Security Applications Conference*, pp.441–450 (2009).

[20] Omote, Y., Chubachi, Y., Shinagawa, T., Kitamura, T., Eiraku, H. and Matsubara, K.: Hypervisor-based Background Encryption, *Proc. 27th ACM Symposium on Applied Computing*, pp.1829–1836 (2012).

[21] Oyama, Y., Giang, T.T.D., Chubachi, Y., Shinagawa, T. and Kato, K.: Detecting Malware Signatures in a Thin Hypervisor, *Proc. 27th ACM Symposium on Applied Computing*, pp.1807–1814 (2012).

[22] Paleari, R., Martignoni, L., Roglia, G.F. and Bruschi, D.: A Fistful of Red-Pills: How to Automatically Generate Procedures to Detect CPU Emulators, *Proc. 3rd USENIX Workshop on Offensive Technologies* (2009).

[23] Raffetseder, T., Kruegel, C. and Kirda, E.: Detecting System Emulators, *Proc. 10th Information Security Conference*, Lecture Notes in Computer Science, Vol.4779, pp.1–18 (2007).

[24] SDBOT, available from ⟨http://about-threats.trendmicro.com/us/malware/sdbot⟩.

[25] Shinagawa, T., Eiraku, H., Tanimoto, K., Omote, K., Hasegawa, S., Horie, T., Hirano, M., Kourai, K., Oyama, Y., Kawai, E., Kono, K., Chiba, S., Shinjo, Y. and Kato, K.: BitVisor: A Thin Hypervisor for Enforcing I/O Device Security, *Proc. 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (*VEE 2009*), pp.121–130 (2009).

[26] Skoll, D.F.: virtdetect, available from ⟨http://search.cpan.org/dist/Sys-Detect-Virtualization/script/virtdetect⟩.

[27] slabbed-or-not, available from ⟨https://github.com/kaniini/slabbed-or-not⟩.

[28] Sun, L., Ebringer, T. and Boztas, S.: An automatic anti-anti-VMware technique applicable for multi-stage packed malware, *Proc. 3rd International Conference on Malicious and Unwanted Software*, pp.17–23 (2008).

[29] The OpenZFS Project, available from ⟨http://www.open-zfs.org/⟩.

[30] virt-what, available from ⟨http://people.redhat.com/˜rjones/virt-what/⟩.

Yoshihiro Oyama received his M.S. and Ph.D. degrees in information science from the University of Tokyo, Tokyo, Japan, in 1998 and 2001, respectively. He has been an associate professor at the University of Electro-Communications, Tokyo, Japan, since 2006.

Yudai Kawasaki received his M.S. degree in informatics from the University of Electro-Communications, Tokyo, Japan, in 2014.

Kazushi Takahashi received his M.S. and Ph.D. degrees in information science and technology from the University of Tokyo, Tokyo, Japan, in 2010 and 2013, respectively. He has been a research associate at the University of Electro-Communications, Tokyo, Japan, since 2013.