CloudVMI: Virtual Machine Introspection as a Cloud Service

Hyun-wook Baek*, Abhinav Srivastava† and Jacobus Van der Merwe*
*University of Utah
Email: baekhw@cs.utah.edu, kobus@cs.utah.edu
†AT&T Labs - Research
Email: abhinav@research.att.com

***Abstract*—Virtual machine introspection (VMI) is a mechanism that allows indirect inspection and manipulation of the state of virtual machines. The indirection of this approach offers attractive isolation properties that has resulted in a variety of VMI-based applications dealing with security, performance, and debugging in virtual machine environments. Because it requires privileged access to the virtual machine monitor, VMI functionality is unfortunately not available to cloud users on public cloud platforms. In this paper, we present our work on the CloudVMI architecture to address this concern. CloudVMI virtualizes the VMI interface and makes it available as-a-service in a cloud environment. Because it allows introspection of users' VMs running on arbitrary physical machines in a cloud environment, our VMI-as-a-service abstraction allows a new class of cloud-centric VMI applications to be developed. We present the design and implementation of CloudVMI in the Xen hypervisor environment. We evaluate our implementation using a number of VMI applications, including a simple application that illustrates the cross-physical machine capabilities of CloudVMI.**

I. Introduction

The flexibility and ease of management that can be realized by virtual machine (VM) technologies have made it a mainstay in modern data centers and enabled the realization of cloud computing platforms. The clean abstraction offered by virtualized architectures has also given rise to the concept of virtual machine introspection (VMI) [1]. VMI allows visibility into and control of the state of a running virtual machine by software running outside of the virtual machine. The indirection offered by this approach is quite attractive, especially for security related applications, and as a result a variety of VMI tools have been developed [1], [2], [3]. Because they are allowed access to the system memory, VMI tools typically run in the privileged domain (e.g., dom0 in the Xen architecture).

LibVMI is an open source implementation of VMI supporting commodity hypervisors such as Xen and KVM [4]. LibVMI provides the functionality of mapping raw memory pages of VMs inside the privileged VM and relies on monitoring software to interpret the contents of these mapped pages. Due to the semantic gap between the information present in the memory pages (raw bits) and the higher-level information used by applications and the operating system (e.g., data structures, files), monitoring software builds the higher-level abstractions by using knowledge of the internals of application or operating system running inside the VM [5]. For this reason, VMI-based monitoring software gets closely tied with the specific operating system, application type, version, or distribution.

Developing and executing VMI-based monitoring applications in public clouds is more challenging compared to self-virtualized data centers. In a self-virtualized data center, customers have complete control over the privileged VM. Unfortunately, users in a public cloud computing platform do not have access to the privileged domain which is under the control of cloud providers. As a result, cloud users cannot deploy or configure custom VMI tools without the help of cloud providers. Given that cloud platforms are the predominant manner in which customers use VMs, this is a serious impediment. From a cloud provider's perspective, offering VMI tools to customers is painstaking. Given that there are so many operating systems, versions, or distributions that can be executed inside VMs, supporting all of them are neither feasible nor a scalable service for providers.

In this paper, we present our work on CloudVMI to address these concerns. CloudVMI allows VMI to be offered as a cloud service by a cloud provider to cloud customers. Specifically, CloudVMI splits the VMI tool development process into two components. The first component that implements the LibVMI functionality to map memory pages is implemented by the cloud provider. The second component that builds the higher-level semantics from the lower-level information present on mapped pages is implemented by the customer. The focus of the work presented in this paper is to look into the first part of how cloud providers will allow users to perform privileged memory mapping functionality.

In essence, CloudVMI allows cloud users to request introspection privileges for their VMs running on the cloud platform. This design solves the aforementioned problems: (i) By virtualizing the LibVMI interface and allowing cloud customers to safely invoke it to request introspection obviates the need of having access to the privileged VM. (ii) Requiring customers to bring their own policies and implementation of monitoring software relieves cloud providers from supporting many application and OS distributions and versions.

We present the design and implementation of the CloudVMI architecture. CloudVMI virtualizes the LibVMI interface and presents the resulting *vlibVMI* to customers' VMI-based tools in a cloud environment. Invocations to vlibVMI are multiplexed and policed according to customer information to ensure that introspection actions are constrained to the cloud resources allocated to the respective cloud user. The vlibVMI interface can be remotely invoked. This functionality enables novel cross-physical-machine introspection capabilities, allowing cloud-centric (as opposed to device-centric) VMI applications. As expected, our evaluation shows a performance impact associated with virtualization and specifically, remote invocations. Nonetheless, our results show adequate performance to facilitate a large class of VMI-based applications. Our implementation utilizes Xen hypervisor-based clouds, however, our approach is general and equally applicable to other VMMs such as KVM.

To summarize, we make the following contributions:

- Realizing the drawbacks of existing VMI design, we design the CloudVMI architecture to allow VMI functionality to be offered as-a-service by a cloud provider.
- We split the VMI tool design process into two components, allowing cloud providers to offer only privileged memory page mapping functionality and requiring customers to bring their own customized monitoring software.
- We present the prototype implementation of CloudVMI using the Xen hypervisor and LibVMI. Our evaluation demonstrates the efficacy of the system by developing a cloud-centric VMI application which utilizes the cross-physical machine introspection capabilities of CloudVMI.

II. CloudVMI

The primary goal of our work is to enable cloud operators to provide VMI capabilities as a cloud service, to enable their users to make use of VMI-based tools to monitor their own VMs, without compromising the security of other cloud users. With CloudVMI we achieve this goal by virtualizing the low level LibVMI interface and exposing the virtualized interface as-a-service in a cloud environment. Below we first describe the use of CloudVMI in such a cloud environment, before describing the virtualized CloudVMI architecture.

A. VMI as a Service

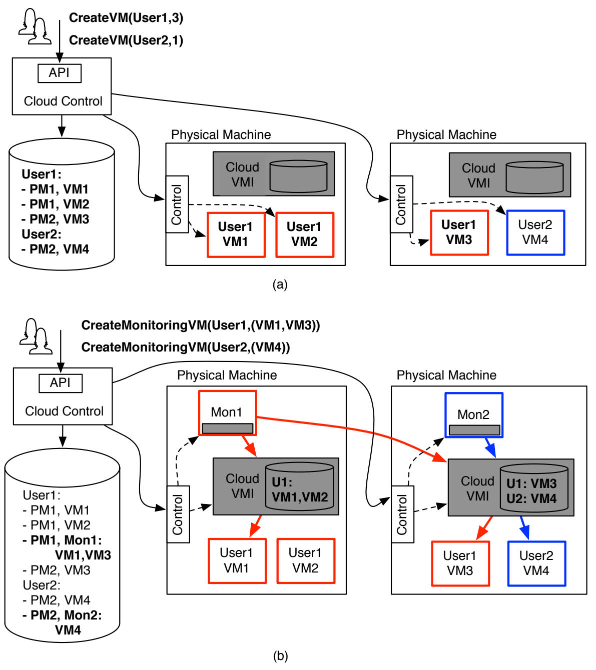

Fig. 1. CloudVMI: VMI as a Cloud Service

Figure 1 depicts a somewhat simplified cloud architecture to illustrate how we achieve the above mentioned goals with CloudVMI. Figure 1 (a) shows two physical machines with CloudVMI capabilities, a simple cloud control architecture and two cloud users invoking the cloud control application programming interface (API) to request the instantiation of three and one VMs, respectively. (To simplify exposition we assume a simple $CreateVM()$ API call and further assume that VMs are named (numbered) from a system wide name space.) Figure 1 (a) shows the fact that the cloud control architecture maintains database entries about users, their virtual resources (VMs) and the physical machines those resources are realized on.

Given this setup, Figure 1 (b) shows the interactions involved with users requesting VMI monitoring capabilities for their previously instantiated VMs (using the $CreateMonitoringVM()$ function). Specifically, $User1$ requests monitoring of $VM1$ and $VM3$ (but not $VM2$), while $User2$ requests the monitoring of its single VM instance, $VM4$. Based on these requests, the cloud control architecture will instantiate monitoring VMs for the users ($Mon1$ and $Mon2$) respectively, and further configure the policy databases on the CloudVMI instances on the relevant physical machines with information needed to perform the necessary multiplexing/demultiplexing and policing. Note that in our simple example $User1$ requested a single monitoring VM to monitor two of its VMs ($VM1$ and $VM3$) that have been instantiated on two separate physical machines. This implies the ability for CloudVMI to allow remote monitoring of VMs. I.e., as shown in the figure, $Mon1$ is used to monitor $VM3$ even though the latter is located on a different physical machine. CloudVMI enables this new remote VMI capability, which will allow a new class of VMI-based cloud-centric applications that can perform VMI functions from a centralized perspective on a set of distributed VMs.

B. CloudVMI Architecture

We describe the CloudVMI architecture in the context of the Xen VMM based environment. The CloudVMI architecture is depicted in Figure 2. As shown in the figure there are two sets of user VMs. *VMs being monitored* are shown on the left, while the *monitoring VMs* are shown on the right.

vlibVMI: CloudVMI virtualizes the LibVMI interface and exposes that via the *vlibVMI* interface. This interface is realized through a combination of the *vlibVMI Server Module* executing in the privileged dom0 and the *vlibVMI Client Library* that executes on each of the monitoring VMs.

The vlibVMI Client Library provides a “LibVMI-like” library against which VMI applications link to invoke the vlibVMI server API. Our initial approach is to have the vlibVMI library mimic the original LibVMI interface as closely as possible to simplify porting of VMI applications developed for LibVMI. However, as we discuss later, we expect the vlibVMI interface to evolve over time to better exploit the fact that CloudVMI offers VMI capabilities across multiple VMs and multiple physical machines in a cloud environment.

As shown in Figure 2, the vlibVMI Server Module performs multiplexing and policing of monitoring VM invocations to ensure that such invocations are constrained to monitored

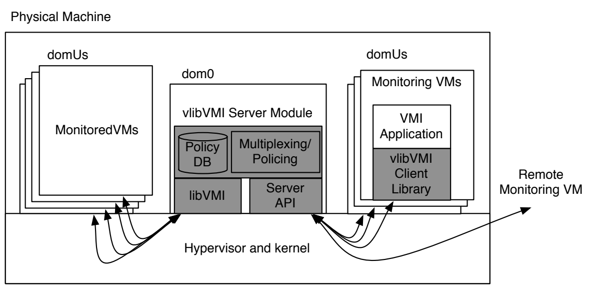

Fig. 2. CloudVMI Architecture

VMs that are associated with the requesting cloud user (or account). As described in Section II-A these actions are driven from a policy database that the cloud control architecture maintains. "Under the hood" the Server Module uses existing LibVMI [4] functionality to introspect monitored VMs. Figure 2 also illustrates the fact that CloudVMI allows VMs in the local physical machine to be monitored by VMI applications executing in *remote monitoring VMs*. This is enabled by exporting the vlibVMI interface as a server API that can be invoked remotely.

State Management: For the original LibVMI and its applications, a *VMI instance* is typically created, which serves as an application-level handle to refer to the state maintained by the underlying library. This VMI instance is used by the application to perform introspection actions, typically across multiple successive introspection requests. Dealing with this state creation and maintenance requires special attention in CloudVMI.

In CloudVMI, the actual state associated with accessing VM memory pages is maintained in the vlibVMI Server Module, while the *client handle* referencing that state is exposed to VMI applications running on the monitoring VMs via the vlibVMI Client Library. VMI applications that do not cleanly terminate and release state associated with its instance handles might result in stale state and memory leakage on the server side. A naive solution for this would be to encapsulate each VMI API call with a pair of VMI instance create and destroy functions. With such an approach every LibVMI function creates a VMI instance first, performs the desired VMI function and destroys the VMI instance. The problem with this solution is that it incurs the cost of VMI creation and destruction with every LibVMI function call. Further, VMI instance creation is relatively heavy-weight compared to other VMI functions: VMI instance creation takes about ten milliseconds, which is three to four orders of magnitude longer than invocation of other VMI functions.

Instead of this secure-but-costly solution, CloudVMI manages a hash table for VMI instances and performs "garbage-collection" for those that are not used for a certain time. Figure 3 illustrates how CloudVMI manages this hash table. CloudVMI maintains a hashtable for VMI instances, and each entry consists of a VMI instance address, a key, the age of the entry since it was used for the last time and the parameters used for its creation and modification. When a user requests to create a new VMI instance, the server creates the instance, pushes it into the hashtable and returns its key (Figure 3, flow 1 to flow 4). Then, the user can use the key to access the VMI instance as they use the pointer value of VMI instance with

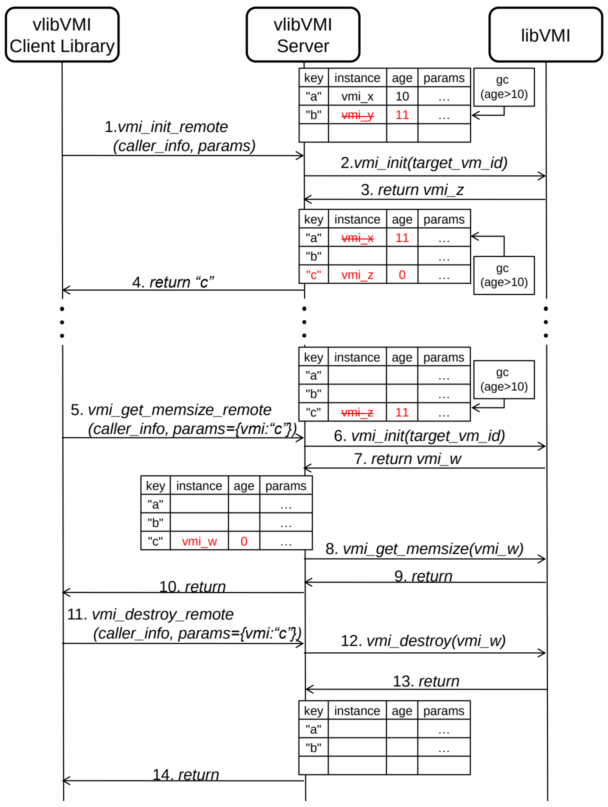

Fig. 3. vlibVMI Server Module state management

the original LibVMI. The indirection provided by this approach has the additional security benefit that actual memory location information is not revealed to the monitoring applications.

A garbage collector periodically checks the ages of VMI instances since it was last used. VMI instances that have not been accessed for a specified time are destroyed by the garbage collector. Maintaining the hashtable entry involves a limited amount of state. Therefore, the garbage collector does not delete the hashtable entry, but maintains its key value and parameter information used for creation and modification. The information is used when the client who created the VMI instance requests to use the VMI instance again. When the server receives a VMI invocation on a garbage-collected VMI instance, the server implicitly recreate the VMI instance based on the previously used parameters. (Figure 3, flows 6 and 7.) This happens only for the VMI instances that are implicitly destroyed - if a user explicitly destroys a VMI instance, the server removes not only the VMI instance, but also its entry in the hashtable (Figure 3, flows 11 to 14). Finally, to prevent memory leakage by forgotten hashtable entries, the Service Module limits the number of VMI instance entries that each user can have. vlibVMI offers an additional function for this feature. CloudVMI users can check the number of VMI instance entries they have and their keys, and they can remove unused entries.

III. Implementation

Our CloudVMI implementation uses Xen and LibVMI. The current version is running on Xen 3.0 and uses LibVMI version 0.9_alpha. As described in Section II-B, CloudVMI consists of two modules, the vlibVMI Client Library and the vlibVMI

Service Module. In our current implementation, the vlibVMI interface is realized using the Linux RPC. The client library makes RPC calls to the privileged virtual machine hosting the Service Module for the target virtual machine to be monitored and returns results to the VMI applications. The vlibVMI Service Module is a server program that receives VMI requests from monitoring virtual machines, performs VMI using the underlying LibVMI interface and returns the results. Our vlibVMI Service Module is implemented in C++. For the current version of CloudVMI, garbage collection, aging and VMI instance entry managing functions are under implementation, and the hashtable manages only VMI instances and their key values.

In addition to the Linux RPC implementation described above, we have a CloudVMI implementation which exposes the vlibVMI interface as a RESTful API. This alternative enables users to perform VMI using HTTP request so that users can introspect their virtual machines regardless of the languages and platform they use. Since it uses HTTP, this implementation has performance limitations, however, we envision that the ease of use of this approach will enable new uses of VMI. Our RESTful implementation is based on Python Flask and the Tornado server.

IV. Evaluation

In this section we present our evaluation of the CloudVMI prototype implementation. Our evaluation was performed in the Emulab[6] testbed environment using a pair of Linux physical machines interconnected via a 100 Mbps link. Each physical machines was equipped with 12-Gigabyte RAM and a 64-bit Intel Quad Core Xeon E5530 CPU. Both nodes were running Ubuntu 12.04 Linux with Xen 3.0. We used Ubuntu 12.04 Linux 3.8.4 for the VMs in our evaluation.

Below we present three evaluation results. First, Section IV-A describes a simple application that we developed to illustrate the cross-physical-machine introspection capabilities provided by CloudVMI. Section IV-B presents a macro-level evaluation of CloudVMI by comparing the performance of a number of simple VMI applications using CloudVMI with the same applications using the original LibVMI functionality. This evaluation also demonstrates the ease with which applications developed to use the original LibVMI can be ported to CloudVMI. Finally, in Section IV-C we present a detailed micro-benchmark evaluation of our prototype implementation.

A. *Cross-physical-machine VMI*

A unique feature of CloudVMI is that it can introspect virtual machines running on remote physical machines. The target monitored VM can be either a single virtual machine or multiple machines spread throughout a cloud infrastructure, allowing the centralized monitoring of virtual machine instances in a cloud environment. Moreover, compatibility with the original LibVMI interface will allow existing VMI-based applications to be readily ported and deployed in a cloud environment.

As an illustrative use case of this functionality, we have modified the *module-list* example distributed with LibVMI to show the list of installed kernel modules in several virtual machines located in different physical machines. This example illustrates the scenario where multiple virtual machines are associated with a single cloud user (or account) and the application monitors the kernel modules associated with all VMs from a centralized monitoring applications. Such functionality might for example be a building block in a cloud security monitoring or cloud forensic application.

B. *Macro-benchmark*

TABLE I. Application Average Running Times

	LibVMI	Cross-VM CloudVMI	Cross-PM CloudVMI
module-list	23.236 ms	24.581 ms	26.021 ms
process-list	22.653 ms	55.647 ms	71.426 ms

To evaluate the overall performance of the CloudVMI, we first ran two existing LibVMI examples on CloudVMI and the original LibVMI and compared the result. The examples we used here are *process-list.c* and *module-list.c* which are distributed with LibVMI. Each of these examples is a console-based application that shows the list of installed kernel modules or the list of running processes of the target virtual machine.

We ran these two examples under three different scenarios: (i) using the original-LibVMI, (ii) using CloudVMI in a cross-VM setup on the same physical machine and (iii) using CloudVMI in a cross physical machine (cross-PM) setup. Specifically, for the original LibVMI case, we ran the examples on the dom0 and introspected a customer VM (domU in Xen's terminology) in the same physical machine. For the cross-VM case, we ran the examples in a domU and introspected another domU within the same physical machine. For the cross-PM case, we ran the examples on a domU and introspected another domU in a different physical machine. With reference to Figure 1 (b), the Cross-VM setup is the case where *Mon1* introspects *VM1*, and the Cross-PM case is when *Mon1* introspects *VM3*.

Table I shows the runtime for each application, averaged over a thousand runs. The target (monitored) virtual machine for this evaluation was a Ubuntu 12.04 system running with 3 kernel modules and 53 processes. In this "static" setup, the number of LibVMI API function calls invoked by each example is fixed. Specifically, the number of LibVMI API function calls were 19 for *module-list* and 173 for *process-list*. This explains why *process-list* takes significant longer time to finish than *module-list* in the CloudVMI cases. More importantly, however, these results suggest that the serial nature of the original LibVMI API might need to be refactored to better suit a cloud-centric VMI.

The performance is of course also sensitive to the size of request parameters and the resulting return values. The API functions called by *module-list* have fewer parameters and small return values (i.e., primitive variable type) compared to *process-list*. Naturally, heavier API functions like *vmi_read_pa( )*, which returns specified size of physical memory in binary will introduce more significant overhead.

C. *Micro-benchmark*

To evaluate the performance of CloudVMI in more detail, we have measured invocation times for several different

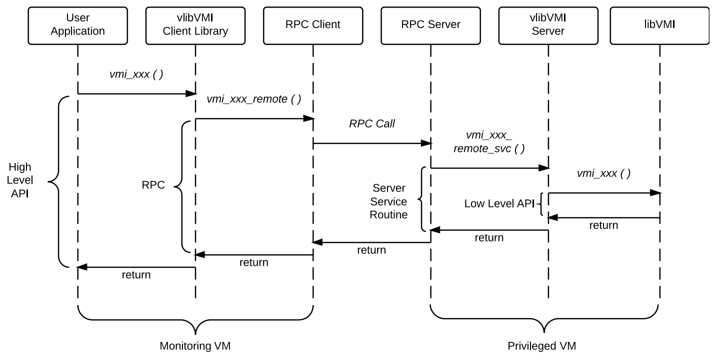

Fig. 4. Invocation Time Sections in CloudVMI

API functions: *vmi_init(), vmi_destroy(), vmi_get_memsize(), vmi_pause_vm() and vmi_resume_vm()*. For each function, we have called it hundred thousand times and calculated the average invocation times. As shown in Figure 4, the invocation times were measured at four different points in the CloudVMI architecture.

With reference to Figure 4, *high-level API* invocation time means the elapsed time from calling a LibVMI-like function (i.e., vlibVMI function) to its return in the user level application. *RPC section* is the elapsed time from calling a Linux RPC function to its return in vlibVMI library code. *Server Service Routine* invocation time is the time from the start of the mapped service routine for the API function to its return. Finally, *low-level API* invocation time is the time taken to perform the original LibVMI function in the server.

TABLE II. INVOCATION TIMES OF LIBVMI FUNCTIONS

	Low-level API calling
vmi_init & vmi_destroy	11456.07 μs
vmi_get_memsize	0.64 μs
vmi_pause_vm	1.79 μs
vmi_resume_vm	1.66 μs

TABLE III. INVOCATION TIMES OF CLOUDVMI FUNCTIONS UNDER CROSS-VM SET-UP

	High-level API calling	Low-level API calling
vmi_init & vmi_destroy	11403.43 μs	11255.80 μs
vmi_get_memsize	60.32 μs	0.77 μs
vmi_pause_vm	59.98 μs	3.53 μs
vmi_resume_vm	58.97 μs	3.22 μs

TABLE IV. INVOCATION TIMES OF CLOUDVMI FUNCTIONS UNDER CROSS-PM SET-UP

	high-level API calling	low-level API calling
vmi_init & vmi_destroy	11647.26 μs	11267.40 μs
vmi_get_memsize	180.17 μs	1.13 μs
vmi_pause_vm	182.98 μs	5.68 μs
vmi_resume_vm	185.35 μs	5.05 μs

As with the macro evaluation in section IV-B, this evaluation has been performed under three different settings, namely original LibVMI, cross-VM (same physical machine) and cross-PM (cross-physical machine).

Table II, III, and IV show the results at the user-level. For the original LibVMI case, it is the time for performing the original LibVMI functions ("Low-level" API invocation time), and, for CloudVMI cases, it is the "High-level" API invocation time. We also measured the times for the original LibVMI functions during each evaluation and present this together with the result of interest for reference. Each entry is the average invocation time to perform a LibVMI-like (or LibVMI) function. For *vmi_init()* and *vmi_destroy()*, we have paired them and measured the invocation time together.

Here, we can see the overhead added by CloudVMI to the original LibVMI functionality. The overhead varies depending on the data size of parameters and return values sent and received, but it is around 60 microseconds for cross-VM case and around 180 microseconds for cross-PM case. Note that the entries for *vmi_init()* and *vmi_destroy()* have doubled overhead because the entries are the sum of invocation times of the two functions. Also, the parameter and return value size of *vmi_init()* is relatively larger than other API functions.

TABLE V. DETAILED INVOCATION TIMES FOR CROSS-VM SET-UP

	vmi_gem_memsize( )	vmi_pause_vm( )	vmi_resume_vm( )
Low-level API	0.77 μs	3.53 μs	3.22 μs
Service Routine	3.71 μs	6.19 μs	5.91 μs
RPC	58.72 μs	58.57 μs	57.57 μs
High-level API	60.32 μs	59.98 μs	58.97 μs

Table V shows more detailed performance information. We note that the major overhead is due to the network interaction associated with the RPC. The average overhead attributable to RPC is about 50 microseconds. We verified that this overhead corresponds to the inherent overhead associated with Linux RPC in our evaluation environment.

TABLE VI. INVOCATION TIMES OF RESTFUL CLOUDVMI FUNCTIONS UNDER CROSS-VM SET-UP

	high-level API calling	low-level API calling
vmi_get_memsize	1105.68 μs	2.50 μs

As mentioned in Section III, CloudVMI exports the vlibVMI interface as a RESTful API. Table VI shows an example of the performance of the RESTful API. This result can be compared with the *vmi_get_memsize* entries in Table II and III. As might be expected, the RESTful API introduces far more overhead than API using RPC. The overhead comes from the many different layers of a RESTful API realization: A request is encoded in JSON, transmitted via HTTP, decoded at the server and passed to a Python server before the original LibVMI interface is invoked. Likewise, results return via the reverse path.

V. RELATED WORK

Monitoring virtual machines has been an active area of research. Researchers have proposed various solutions to monitor VMs for attack detection [7], [8], [9], [10], malware analysis [11], [12], [13], debugging [14], and performance [15]. Garfinkel et al. [1] developed a system called Livewire that uses VMI techniques to inspect the runtime security state of virtual machines. Payne et al. [3] created a similar system named XenAccess for VMs running atop the Xen hypervisor [16]. Later, the XenAccess introspection library was ported to other VMMs such as KVM and named LibVMI. Realizing the efficacy of VMI techniques, many security applications using VMI have been developed. Srivastava et al. [2] proposed a white-list based application-level firewall for the virtualized environment that performs introspection for each new connection to identify the process bound to the connection. Ziang et al. [17] developed out-of-the-box security approach to detect attacks ocuring inside a VM. While these approaches have demonstrated the usefulness of VMI, they were developed for the self-hosted virtualized environment and did not take the complexity of elastic computing environment such as clouds into account. In contrast, with CloudVMI, we offer VMI-as-a-cloud service, empowering cloud users to develop their own customized monitoring applications. Further, existing approaches of VMI was device-centric, i.e., limited to the VMs on a single physical machine. In contrast, CloudVMI allows cloud-centric introspection by enabling virtual machine introspection across physical machines.

Realizing that the security concern is the main obstacle in the wider adoption of cloud computing, researchers have proposed various ways to offer security-as-a-service in the cloud. Srivastava et al. [18] have proposed the notion of cloud app market that describes the delivery of security solutions via apps similar to smartphone apps. To expose hypervisor-level functionality to cloud users, they suggested modifications to virtual machine monitors and/or nested virtualization. Butt et al. [19] created a self-service cloud platform that allows cloud customers to flexibly deploy their own security solutions. Srivastava et al. [20] created a trusted VM snapshot generation service in the cloud that operates even if cloud administators are malicious. Brown et al. [21] offered trusted platform-as-a-service for cloud customers to deploy applications in the cloud in a trustworthy manner. Similar to these efforts, we envision VMI-as-a-Service in the cloud to allow cloud users to develop security monitoring applications customized for their VMs.

VI. Conclusion and Future Work

We presented our work on CloudVMI which allows virtual machine introspection to be offered as-a-service by cloud providers. CloudVMI virtualizes the low level VMI interface and polices access to ensure VMI actions are constrained to resources owned by the respective cloud users. Further, CloudVMI allows VMI actions to be performed across different physical machines, thus allowing for cloud-centric introspection. We presented an evaluation of our proof-of-concept implementation of CloudVMI. Our results prove the feasibility of offering VMI as a cloud service and specifically making VMI cloud-centric compared to the current device-centric approaches. Our results also suggest that compatibility with the existing LibVMI will simplify porting of VMI-based tools to the CloudVMI environment. At the same time our results indicate that cloud-aware extensions of the VMI interface, for example allowing for batching of requests that might simply be serialized in a traditional non-cloud approach, might be beneficial. We plan to explore such extensions as part of our future work. Finally, we plan to focus future work on the second component of our split VMI-tool development approach by developing more sophisticated VMI applications, including cloud-centric VMI applications which can utilize CloudVMI's unique cross-physical-machine capabilities.

References

[1] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *NDSS*, San Diego, CA, February 2003.

[2] A. Srivastava and J. Giffin, "Tamper-resistant, application-aware blocking of malicious network connections," in *RAID*, Boston, MA, September 2008.

[3] B. D. Payne, M. Carbone, and W. Lee, "Secure and flexible monitoring of virtual machines," in *ACSAC*, Miami, FL, December 2007.

[4] "vmitools Virtual machine introspection tools," http://code.google.com/p/vmitools/.

[5] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *Security and Privacy (SP), 2011 IEEE Symposium on*, 2011, pp. 297–312.

[6] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An Integrated Experimental Environment for Distributed Systems and Networks," in *OSDI*, 2002.

[7] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "ReVirt: Enabling intrusion analysis through virtual-machine logging and replay," in *OSDI*, Boston, MA, December 2002.

[8] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.

[9] L. Litty, H. A. Lagar-Cavilla, and D. Lie, "Hypervisor support for identifying covertly executing binaries," in *USENIX Security Symposium*, San Jose, CA, August 2008.

[10] A. Srivastava and J. Giffin, "Automatic discovery of parasitic malware," in *RAID*, Ottawa, Canada, September 2010.

[11] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," in *ACM CCS*, Alexandria, VA, October 2008.

[12] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *ACM CCS*, Arlington, VA, October 2007.

[13] X. Jiang and X. Wang, "Out-of-the-box monitoring of VM-based high-interaction honeypots," in *RAID*, Surfers Paradise, Australia, September 2007.

[14] S. T. King, G. W. Dunlap, and P. M. Chen, "Debugging operating systems with time-traveling virtual machines," in *USENIX Annual Technical Conference*, Anaheim, CA, April 2005.

[15] B. C. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, and R. N. Chang, "vpath: Precise discovery of request processing paths from black-box observations of thread and network activities," in *USENIX Annual Technical Conference*, San Diego, CA, June 2009.

[16] 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 *ACM SOSP*, Bolton Landing, NY, October 2003.

[17] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through VMM-based 'out-of-the-box' semantic view," in *ACM CCS*, Alexandria, VA, November 2007.

[18] A. Srivastava and V. Ganapathy, "Towards a richer model for cloud app markets," in *ACM Cloud Computing Security Workshop*, 2012.

[19] S. Butt, A. Lagar-Cavilla, A. Srivastava, and V. Ganapathy, "Self-service cloud computing," in *ACM CCS*, 2012.

[20] A. Srivastava, H. Raj, J. Giffin, and P. England, "Trusted VM Snapshots in Untrusted Cloud Infrastructures," in *Proceedings of the 15th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2012.

[21] A. Brown and J. Chase, "Trusted platform-as-a-service: A foundation for trustworthy cloud-hosted applications," in *ACM Cloud Computing Security Workshop*, 2011.