Forensic Virtual Machines: Dynamic defence in the Cloud via Introspection

Adrian L. Shaw†, Behzad Bordbar*, John Saxon*
* School of Computer Science
University of Birmingham
Birmingham, UK B15 2TT
Email: { b.bordbar, j.t.saxon, k.harrison } @cs.bham.ac.uk

Keith Harrison*, Chris I. Dalton†
† Hewlett-Packard Laboratories
Longdown Avenue, Stoke Gifford
Bristol, UK BS34 8QZ
Email: { adrian.shaw, cid } @hp.com

***Abstract*—The Cloud attempts to provide its users with automatically scalable platforms to host many applications and operating systems. To allow for quick deployment, they are often homogenised to a few images, restricting the variations used within the Cloud. An exploitable vulnerability stored within an image means that each instance will suffer from it and as a result, an attacker can be sure of a high pay-off for their time. This makes the Cloud a prime target for malicious activities. There is a clear requirement to develop an automated and computationally-inexpensive method of discovering malicious behaviour as soon as it starts, such that remedial action can be adopted before substantial damage is caused. In this paper we propose the use of Mini-OS, a virtualised operating system that uses minimal resources on the Xen virtualisation platform, for analysing the memory space of other guest virtual machines. These detectors, which we call Forensic Virtual Machines (FVMs), are lightweight such that they are inherently computationally cheap to run. Such a small footprint allows the physical host to run numerous instances to find symptoms of malicious behaviour whilst potentially limiting attack vectors. We describe our experience of developing FVMs and how they can be used to complement existing methods to combat malware. We also evaluate them in terms of performance and the resources that they require.**

I. Introduction

Security is often cited as one of the most contentious issues in Cloud computing. It is argued that as the Cloud is intended to handle large amounts of data, attackers can be sure of a high pay-off for their activities. This makes the Cloud a prime target for malicious activities. In addition, to benefit from the economies of scale, the applications and operating systems are homogenised to a few images restricting the variations of products used within the Cloud. As a result, a vulnerability can be exploited on a large number of machines. There is a clear need to develop dynamic defence techniques to discover malicious behaviour as soon as it starts so that remedial action can be taken before substantial damage is done. This paper builds on Harrison et al., which described Forensic Virtual Machines (FVMs) [1]. At the heart of the approach is the observation of a trend in the reuse of malicious *components* by the writers of malware [2]–[4]. The use of components seem to be partly due to pressures to write better quality malware by reusing existing components and algorithms used by other malware writers.

However, reusing components with malware can produce symptoms which are common between not only variants of one piece of malware but also between different kinds of malware. For example, most malware on Windows set one or more registry keys to unusual values [5], [6]. It is also common for malware to abort processes such as anti-virus systems to prevent detection [6]. We view unusual changes to the registry and aborting processes as examples of *symptoms* of malware. Symptoms are different from malicious behaviour; symptoms are detectable traces of activities that facilitate a malicious activity. For example, if malware changes the value of a registry key to an unusual value and we can detect that change, we can interpret this as a potential sign of malicious activity. Of course, registry values can be changed for perfectly legitimate reasons. However, observing unusual changes to the registry at the same time as particular processes disappearing, such as an anti-virus product, must increase our suspicion that the system is infected so that remedial actions can start.

Building on the above observation, FVMs are small Virtual Machines (VM) that can monitor other VMs to discover the symptoms in real-time via Virtual Machine Introspection (VMI) [1]. FVMs are small; each FVM is dedicated to identifying only one symptom. As a result, the crucial part of the code within the FVMs can be manually inspected. In addition, FVMs exchange messages via secure multi-cast channels to share information about the discovery of symptoms within the VMs. This allows the FVMs to conduct distributed monitoring; if an FVM detects a symptom in a virtual machine, it will inform other FVMs to come to its assistance in order to detect the presence of other symptoms. The more symptoms are detected, the more we can be sure of the possibility of malicious behaviour. The FVMs report to a C&C centre that collects and collates the information. The FVMs, C&C and communication channel act as an autonomous system for dynamic defence. The C&C module can use the virtualisation mechanism to "freeze" the VM by denying it any CPU cycles, effectively stopping the malicious activity. The memory will remain frozen until it can be quickly reviewed or alternatively can be copied for a complete forensic analysis.

Harrison et al. described the overall vision of the FVM. In this paper, we report our experience in engineering prototype FVMs for detecting a wide range of symptoms. To keep the FVMs small we have used the para-virtualised Mini-OS operating system as the basis to produce FVMs for Xen [7]. We have also used the FVMs for detecting symptoms associated to ZeuS [5], Spyeye [8] and Gauss [9]. In addition, we have conducted a number of experiments to evaluate the suitability and performance of Mini-OS for introspection. We also discuss

the potential shortcomings of using Xen and Mini-OS and propose solutions to overcome them.

The paper is organised as follows. The next section briefly describes preliminary information about virtualisation, introspection and Mini-OS. In Section III we describes an overview of FVMs, which is followed by an explanation of the problem addressed in the paper. We shall present evidence to support the presented hypothesis. Section V sketches the proposed solution and highlights the architecture of the FVMs followed by Section VI which reflects on the challenges of creating the FVMs. Section VII evaluates the performance of the prototype FVMs. Discussions about related work are in Section VIII and the paper then ends with a conclusion.

II. Preliminaries

In this section we shall briefly review preliminary material which will be used in the rest of the paper.

A. Introspection

Virtual Machine Introspection (VMI) is a technique that enables one guest *virtual machine* (VM) to monitor, analyse and modify the state of another guest VM by observing its virtual memory pages. Introspection is a powerful technique allowing the security related software to inspect VMs while remaining hidden. Despite this, it has been proven possible for malware in a VM to detect whether they are running on a virtualised platform [10], [11]. Additionally it could also be possible for malware to detect that they are being monitored by an FVM through the use of side-channel attacks that rely on shared physical resources such as a vCPU cache [12]. These problems are not within the scope of this paper, but could be mitigated by exclusively allocating FVMs to their own shared CPU core which is kept away from use by customer VMs.

One of the early methods of introspection on one VM from an external VM was by Garfinkel and Rosenblum [13]. They used VMI to develop an Intrusion Detection System (IDS), called Livewire, for a customised version of VMware Workstation for Linux. VMI techniques have also been used in digital forensics [14], [15]. Hyperspector [16] implemented another IDS for distributed systems using VMI to isolate the IDS from the servers that they monitor. These isolated systems are located inside distinct VMs which are called IDS VMs. There have also been commercial products released which have been built using VMI techniques [17].

B. Mini-OS

In our implementation we have used Mini-OS [18]. Mini-OS began as an example from the Xen Community of how to port a kernel to the Xen architecture for para-virtualised applications. We chose Mini-OS because it is an extremely small kernel, with a minimal install resulting in less than 1MB in size. It does not have a userspace or multi-threading capabilities. Other extras can be attached to Mini-OS, such as a TCP/IP stack, a standard C library and a basic graphical system. In addition, there is an experimental network file-system under development. However, with the exception of the standard C library, we did not include such additional functionality in order to reduce the size of the attack surface. An additional advantage of the small size is the ability to run a larger number of FVMs within available infrastructure. For further details about Mini-OS we refer the reader to a a short guide to the kernel startup process [19].

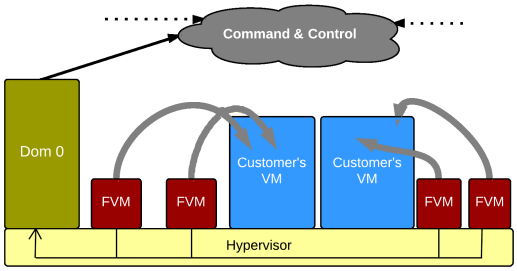


Fig. 1. FVMs inspecting Customers' VMs

III. Forensic Virtual Machine

In this section we shall briefly review the idea of Forensic Virtual Machines [1]. We are aiming to address two problems. Our first aim is to present an approach to detect a diverse set of malware. To do so, we propose looking for the *symptoms* of the malicious behaviour as opposed to looking for the malicious behaviour itself. This is similar to looking at symptoms of disease in the human body in order to identify unhealthy individuals. Secondly, in order to make efficient use of machine resources, our FVMs look for symptoms in multiple virtual machines. This reduces the cost of creation of FVMs. In addition, the sharing of the FVMs between multiple virtual machines results in increasing security, as the attackers cannot know how many FVMs are currently observing them. In Section III-A we will explain the outline of the approach. In Section III-B we describe a wide range of symptoms and their relationship to major security attacks.

A. Introspecting Guest VMs via small VMs

Figure 1 depicts an outline of the approach implemented in this paper. It shows a number of small independent VMs, called Forensic Virtual Machines (FVMs), which have been given the capability to inspect the memory pages of specific customer VMs. Once a symptom has been detected, then the FVM reports its findings to other FVMs. In such cases, other FVMs will be prompted to inspect the VM for the additional symptoms. In addition, when a symptom is discovered, this fact is immediately reported via *Domain 0* (Dom0) to a C&C. The C&C correlates this information with information from other sources to identify an appropriate mitigation. For instance, the C&C, through Dom0 and hypervisor, can "freeze" the customers' VM by denying it any CPU cycles in order to stop the malicious activity. The memory will remain frozen until it can be forensically examined or copied for further analysis.

B. Detecting Symptoms via Introspection

In ordinary life, symptoms of human body illnesses often prompt us to ask for medical help. In our approach detecting symptoms will also help us to single out infected virtual machines among a large number running on a blade.

Taking the malware family ZeuS [5] for example, when it is installed by an administrator account, it alters the `Userinit` registry value in order to automatically load itself again during system start-up. The majority of applications will leave this alone and create a log-on start-up script rather than amending this registry value. A change to this value could very well be considered a potentially malicious act and hence can be counted as a symptom. Another viable symptom would be the use of non-pronounceable mutex names. Developers regularly use human readable keys to make debugging easier. ZeuS v1 uses the string of characters "_AVIRA_x" where x is a number dictating the component in question, whilst the next version (ZeuS v2) uses randomised GUIDs, both of which may be considered as not being pronounceable.

C. Mobility algorithms

FVMs, by design, include a set of distributed algorithms that describe which VM is next for analysis and how long for. These algorithms could range from very simple random inspections to dynamic ones which consolidate the number of current symptoms and their respective types. These not only can introduce a level of unpredictability to an attacker who wishes to anticipate their movements, but can also optimise FVMs to concentrate monitoring resources where they are most needed. We refer to such algorithms as *mobility algorithms*, which can be tailored by the infrastructure provider. In our previous work a sample of a mobility algorithm is presented [1]. In addition, the paper reports on a simulator that we used to study the outcome of such algorithms.

IV. Description of the Problem

In the rest of this paper we shall describe our implementation of the FVMs and report on their performance. The following are four main requirements which were followed.

A. Smaller FVMs to reduce attack surface

It is essential for the FVM to be as small as possible to reduce the attack surface. All non-essential libraries must be removed, drivers which are typically used for general purpose communications have no use in FVMs and must be omitted and efficient coding practices must be adopted. With the omissions in place, there are less malleable elements that can be compromised.

B. Modularised Design to allow reuse

It is essential to reuse the FVM code to ensure that the security experts can examine and certify the reused part *only* once. As a result, the architecture of the FVM must be modularised. Identifying correct modules and creating suitable APIs is highly non-trivial, as the shared modules must be used to write a wide range of symptom detectors and mobility algorithms. Creating modularised FVM allows better testing, as testing can be carried out on individual components focusing on a single aspect of the development at a time.

C. Efficient Introspection

VMI requires computational resources, which could otherwise be allocated to the clients; indeed computation is one of the key commodities provided by the Cloud. Producing a large number of virtual machines to monitor a guest virtual machine can result in a waste of valuable computational resources. Since Cloud systems are expected to be very large, any practical method of inspecting the host VMs must be scalable. As a result, a key crucial challenge is to ensure that searching algorithms are not resource intensive and high in complexity. In Section VII, we shall present the outcome of some measurements on the amount of the resources used by the FVMs, so that they can be allocated appropriately.

D. Lightweight, aiming to remain undetected

It is essential to reduce the complexity of the FVMs to decrease the chance of introducing unwanted side-effects. To do so, the following requirements when introspecting the memory pages are followed:

- *FVMs should only read.* Granting permission to the FVMs to write into the guest memory might jeopardise the integrity and security of the guest domain. In addition, disabling write access will reduce the attack surface and will help in hiding the activities of the FVMs from perpetrator of the malicious activities.
- *Each FVM should only look for one symptom.* This will not only aid in reducing the size and complexity of the FVMs, but also make it easier for manual code review of the searching algorithm used within the FVM.
- *FVMs should introspect one domain at a time.* This will reduce the chance of contamination, allowing the FVMs to perform a simple sensory function. As a result, no local analysis is carried out; the FVM gathers the information and the analysis is carried out at the C&C module.

V. Architecture of an FVM

We have created an architecture comprising four modules as depicted in Figure 2.

A. Main Component

Management of the FVM is handled in the *Main component*. It provides an abstract API to the rest of the FVM for executing a number of functions. This includes posting of messages (*postMessage*) so that the FVM can communicate with other FVMs and the C&C, moving of FVMs from one VM to another (*moveVM*) and *getVMState*, which retrieves information about the symptoms discovered by a given VM. The Main component also checks whether the Time To Live (TTL) has expired, which means that the FVM must move to inspecting another VM. The Main component wires all other components together.

B. Blackboard Inter-domain Communication

The communication between FVMs is implemented via a shared message-board mechanism. FVMs store (multicast) information about their activities such as a list of symptoms discovered for each VM and the last time a VM was visited within a *blackboard*. As depicted in Figure 2, each FVM has

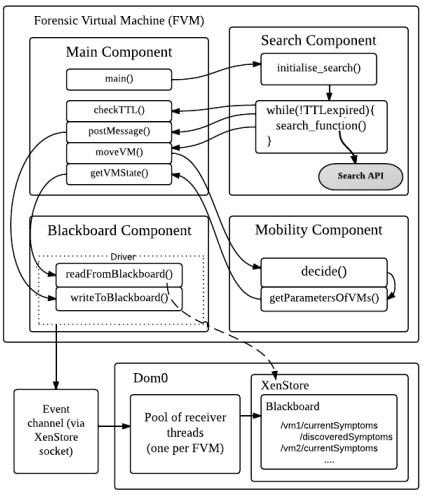


Fig. 2. Architecture of an FVM

a *blackboard component* that incorporates a *driver* for reading and writing from the blackboard. The blackboard itself is kept within Dom0 using XenStore and is shared between multiple FVMs. Each FVM uses an *event channel* via a third party library called XenStoreSocket [20] to interact with Dom0. Within Dom0, we have created a *pool of receiver threads* that process events arriving through the event channel. This architecture allows multiple FVMs to read and write to the blackboard with Dom0. Each interaction is implemented as a transaction.

Messages about the discovery of symptoms are posted on the blackboard. The FVMs also report if no symptoms are discovered. We treat both types of information with equal importance, as the disappearance of a symptom can also be useful for identifying a different type of symptom. However, since FVMs' time for searching for a symptom is often capped, it is possible that the FVM is moved prior to discovery of the symptom. Also, it is possible for an FVM to search the whole memory space of an FVM and yet misses the symptom it is looking for. To explain this assume the scenario that an FVM is scanning the memory space for a given string of bytes. It is possible that malware can alter the parts of memory which have already been inspected. Such newly altered strings of bytes can be discovered only by re-running the FVM or another FVM of the same type.

C. Search Component

The Search component consists of two parts, a *Search API* which is common across all FVMs and a *Search function*. The two parts are wrapped in a piece of code which initialises and executes the Search function. The *s*earch function itself makes use of a *Search API*, which provides primitives for searching memory pages and process tables with the help of the LibVMI library [21] or a wrapper that we have developed for combining functions with the LibVMI library to conduct popular types of search.

Using the Search API, a *s*earch function is composed which implements a searching algorithm. For example, to search for a string in the memory pages of a given process, a searching algorithm can obtain the pages using function two in Table and then use a *s*tring searching algorithm, such as Boyer-Moore [22] or Knuth-Morris-Pratt [23]. The search algorithm is implemented as a function written in the C programming language and compiled to executable code. The control of execution is passed to this executable code once the FVM initialises and has arrived at its target virtual machine. This will allow the reuse of searching algorithms.

Within the FVM Search Component, there is an initialisation function to get the data required to introspect the target VM. This function uses LibVMI initialisation to prepare the FVM for the introspection. This function is depicted as initialise_search() in Figure 2. If the initialisation is successful, then the search_function is executed repeatedly while the TTL has not expired. To do so, the checkTTL function of the Main Component is used for checking TTL and then the *moveVM* function is used for moving from one target VM to another within a predefined neighbourhood of the guest VMs which the FVM is assigned to. In case of discovery or absence of symptoms the function *postMessage* of the Main Component is used to update the blackboard.

D. Mobility

We have implemented three mobility algorithms (see Section III-C). We have implemented a Random Scheme in which the target VM is identified randomly from the VMs within the neighbourhood. We have also implemented a Round-robin scheme that circles through VMs within the neighbourhood one-after-another. Finally, we have implemented the algorithms described in [1]. These schemes are files written in the C programming language which are included in the FVM and the user is able to choose which mobility scheme to use within an FVM by passing it in as a Mini-OS kernel parameter.

When the Time to Live in the Search Component has expired, the moveVM method of the Main Component is invoked, which starts the execution of the $decide()$ method of the Mobility Component. This method executes the mobility scheme which the FVM is initialised with. The scheme requires parameters which are obtained from the Main Component by executing the getParametersOfVMs method. The Main Component obtains this information by reading the contents of the blackboard through the execution of the getVMState method as depicted in Figure 2.

VI. Engineering of the FVMs

In this section we shall discuss the technical aspects of the architecture described previously and its relationship to the requirement of the system described in Section IV.

A. Libraries used and modified

LibVMI is incompatible with Mini-OS, which was chosen to produce small FVMs. It was modified and ported to run on the Mini-OS para-virtualised kernel. Since Mini-OS does not have a GNU LibC, we opted to use the embedded C library known as Newlib [24], which has a minimal set of C libraries to do essential tasks such as I/O, regular expression parsing, string manipulation and producing timestamps. This allowed us to keep the FVM small.

B. Xen Hypervisor and access control

The Xen hypervisor is completely agnostic to the concept of different types of virtual machines (e.g. VM vs FVM). Furthermore, the standard security model prevents a virtual machine, other than Dom0, from looking into the pages of another VM. The access control of Xen was modified to grant the FVMs sufficient privileges to be able to introspect the multiple target virtual machines which define its neighbourhood. In doing so, we allow an FVM to have access to the memory of all required domains. These changes leave the security of ordinary virtual machines unaffected. There is a clear need for research into designing suitable access control mechanisms within Xen to extend the current limited access control model. In particular, any candidate model should differentiate between an FVM and VM in terms of access control. This remains an area for future work.

C. Virtual Machine Introspection via LibVMI

In our earlier work, the XenAccess library was used for introspection [1]. In the current version we make use of the LibVMI library, which supersedes the previous XenAccess project. Suppose that we are interested in establishing whether the address space of a given process, such as Internet Explorer or Firefox, running within a virtualised guest OS, contains a particular text sequence or section of (malicious) machine code. There are four steps to search for such a text sequence or code within the address space of a process running in the target VM.

Step 1: Transferring of VM Introspection meta-data. An FVM needs to access the meta information of a virtual machine target, such as offset values, the name of the VM and the memory size of the VM. This information is typically kept in the XenStore entry for each virtual machine. Access to this information is given to each FVM before they are all launched. It is not desirable to embed the introspection offset information within the FVM for practical reasons. For example, adding a new guest required recompilation of the FVMs. As a result, an entry containing this information is made in the XenStore prior to its creation and configured with read-only access for FVMs alone, no access by the rest. Having VM meta-data entries shared across FVMs helps avoid redundant duplication of this information. Additionally, all the kernel parameters of a Mini-OS stubdomain can only be a total length of 1024 chars, which is probably not long enough for all the introspection information for a whole neighbourhood of VMs. These pieces of data are passed using an FVM launcher script in Dom0.

Step 2: Determining and utilising OS offsets. In this step we locate the offset of the target guest kernel task structures using the parameters from Step 1. For both Windows and Linux, the guest (kernel) virtual address of these structures is either a well-known value or easily determinable. A guest operating system (Windows or Linux for example) maintains such internal structures that describe the application or tasks currently instantiated on the system. Included in the task structure for a particular process is a pointer to a region that contains the page tables that should be loaded when that process is running. Also included in that task structure, is a list of areas of the application's virtual address space that it is actually using. Together, the page tables and virtual address space allow the determination of the actual system physical memory locations being used by that application process.

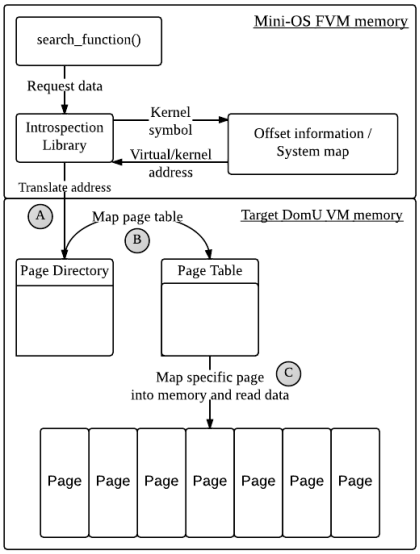


Fig. 3. Virtual Address Translation

Step 3: Inferring correspondence of virtual address to physical address. In this step, we start from a task structure within the guest's virtual address. Then we calculate the machine's physical address for the task structure. Then the physical page at that address is mapped into the FVM. As a result, the FVM can access the contents of the guest's physical page and examines it. It can for example copy the contents locally or analyse directly.

Taking Linux as an example, the page tables of any application process running within the OS contain a mapping for kernel virtual addresses of that OS as well as the application addresses (paging protection mechanisms prevent application processes actually accessing the kernel memory). On x86 based platforms, the root location of the set of page tables from the currently running guest process can be found by examining a particular system register (CR3). This process is depicted in (A) of Figure 3. This register machine state is available to the FVM from the Hypervisor (Xen). Given that register value (which is actually a physical memory address), the FVM maps the physical page containing that physical address into its own address space, see (C) of Figure 3. From that, it can traverse the page tables being used by the target guest OS (mapping additional physical pages from the guest as required) until it finds the physical page corresponding to the virtual address of the guest OS task structures.

Step 4: Crawl through the page tables. After identifying the task structure of the guest OS, we can locate the actual page tables and memory regions used by the process which the search_function is applied to. By proceeding along similar lines as step (B) including page table traversing, the FVM can now map the physical memory actually in use by the application process of interest into its own address space and inspect the contents.

D. FVM Message Format

Each symptom message includes a unique globally identifiable *messageID*. As each FVM looks for a unique type of symptom, we have included *Type* to capture the unique corresponding symptom of an FVM. The message is posted by an FVM with the identifier *FVM-DOMID* which is introspecting a VM identified globally by *TargetDomID*. There are also a few reserved identifiers which are used for miscellaneous message information. For example, a message could be sent to indicate that the FVM has moved and is about to begin introspection on a new domain. Another possibility is that a message could be sent to inform the monitor that it is about to die cleanly (or, if possible, that a fatal accident occurred). New message entries are sent to the blackboard location, which only FVMs have access to.

There was a choice of either using a logical clock system or to rely on the underlying time functionality on the Mini-OS. The problem with using logical clocks is the overhead of synchronisation, a complication that FVMs should not have to deal with. Thankfully, since the Mini-OS uses para-virtualised interfaces, its time library can use a direct interface to the clock that the Xen hypervisor provides, which proves to be sufficiently reliable for research purposes.

E. Blackboard Inter-domain Communication

We have delegated the task of writing to the blackboard to Dom0 to enforce mutual exclusion during updates. Each FVM creates a direct connection to a backend daemon in Dom0, which listens for symptom messages. This is achieved through the use of Xen event channels. Due to the modularity of our design, it is easy to switch the blackboard controller in the FVM for another, such as one which uses the Xway inter-domain communication library [25].

VII. Evaluation

In order to illustrate the suitability of Mini-OS as a viable operating system for FVMs, we evaluate its performance in completing tasks that will be commonplace, for instance accessing the process table. In our experiments we have used a HP EliteBook 8540w with a dual core Intel i7, 8GB of main memory and with hyperthreading disabled was used to perform the tests. It had installed the Xen 4.1.2 hypervisor with Dom0 running Ubuntu 12.04, Xorg Server, Blackbox and the Xen management tools. Dom0 was allocated 1.5GB of memory with the remainder being available to other domains. Each FVM uses at most 4MB of memory. In the future we intend to reduce the size of FVMs; however this will have to be changed within Xen. The Xen hypervisor generally only uses large pages to prevent context switching when navigating often large VM allocated memory. With all tests, the target VM was an idle hardware virtualised domain running Windows XP with 300MB of main memory and a single vCPU.

Profiling: The granularity of timing measurements is dependent on the `gettimeofday` function within Mini-OS. The wall-clock is then updated frequently by the Xen hypervisor, so some level of inaccuracy is expected. However since Mini-OS is a paravirtualised guest, the `gettimeofday` clock has less synchronisation overhead, and is better suited for this purpose than that of the equivalent function within a hardware virtualised domain (HVM). Once the tests were completed, this data was passed from the FVM to a receiver daemon on Dom0 via the Blackboard mechanism.

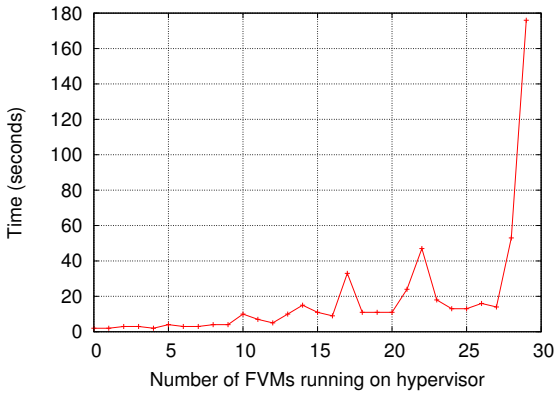


Fig. 4. Time to spawn a new FVM.

Scheduling: The credit-based scheduler that comes with the Xen 4.1.2 source treats all VMs as equal for vCPU time, including both FVMs and VMs. This lack of distinction between a VM and FVM requires a new scheduler to better handle the case of FVMs, otherwise FVMs may starve the client VM of vCPU cycles as they will run at 100%. We have reserved this for future work and do not deeply assess scheduling and scalability in this paper.

A. Startup

When an FVM boots Xen must start the Mini-OS kernel and then the FVM gathers information about its target.

1) Mini-OS Boot: When Xen creates a DomU it must start internal process to allocate memory, access permissions and the necessary components, such as the vCPU. This will become more time consuming the more VMs it starts. Here we test how long it takes to create Mini-OS DomUs. Management of domains on Dom0 was controlled using the XM management tools, which interact with the backend Xend daemon. These tools spawn domains one at a time; in order to maintain a consistent workload of FVMs we made them continue to execute until a particular time.

In Figure 4 we see the time in seconds to spawn a new FVM given a number of existing and working FVMs. We can see that as the number of FVMs increases, the speed at which Xend spawns new domains reduces. The spikes in the graph are likely to be the behaviour of the scheduler readjusting to provide Dom0 more execution time. This test shows that the spawn time is quite significant if there are already a large number of FVMs running on the hypervisor. This could very well be changed with a new scheduler, allowing Dom0 and client VMs to be a higher priority than FVMs.

2) FVM Initialisation: Once Mini-OS has booted, the FVM must acquire information regarding its target. This process includes gathering introspection meta-information about the target VM from the XenStore, determining the memory layout and using this information to find the kernel symbol table within memory. This act can occur numerous times in the lifecycle of the FVM instance as it moves from VM to VM.

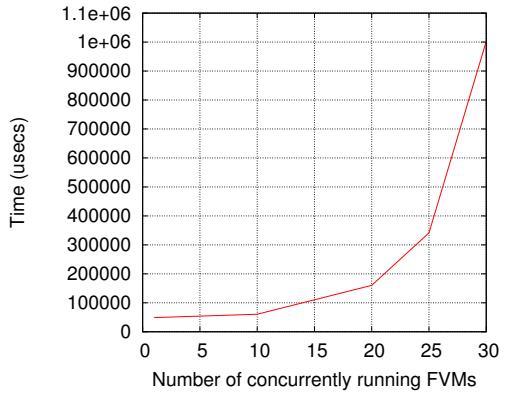


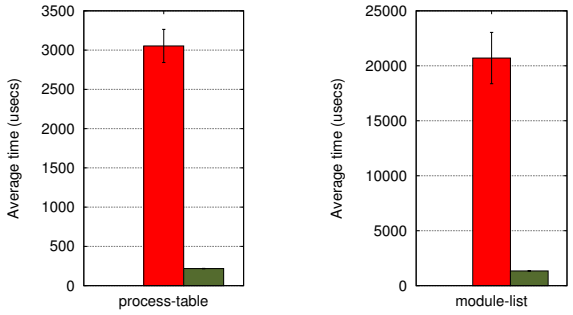
Fig. 5. Mean time to initialise VMI meta-data.

Each FVM reported the mean time taken, in microseconds, to initialise the VMI meta-data ten times, and the results were then averaged once more with the number of concurrent instances running at that particular time. Figure 5 illustrates this. When a single FVM was initialised, the mean average was 48873μs. The exponential behaviour of this is due to the delay between the FVMs and XenStore on Dom0. This bottleneck could be substantially mitigated by passing all of the information it requires as kernel parameters; however this does not allow for updates or additional virtual machines. Another alternative to reduce the memory footprint would be to host the information on distributed high performance domains also running Mini-OS to provide the latest data to the FVMs.

B. VMI Techniques

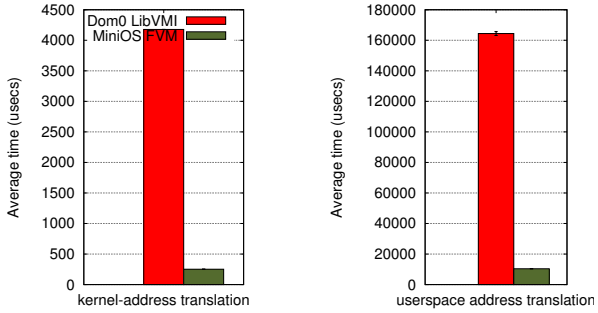
Each FVM will complete numerous tasks in order to navigate the memory of a VM, including virtual address translation in kernel or process space. In order to test the performance of our FVMs over the alternative of using VMI on Dom0, we compare the throughput of applications in both settings. The first application attempts to access the process table and obtain all of the running process names, seventeen in total, on the target host. Figure 6(a) shows that the mean average of getting access to this data was 217μs from our FVMs as opposed to 3116μs when run on Dom0. An additional test was made to loop through the modules available to the Windows guest. The target had eighty five loaded modules and Figure 6(b) shows it took 1339μs from the FVM opposed to 20706μs on Dom0. These particular tests were chosen as they have a similar function to iterating a list within kernel memory.

A significant proportion of execution time involves translating virtual addresses and mapping the data from the target VM. In order to stress-test this, we chose to access the first one hundred bytes of a process, individually, in both kernel and user space. In order to decrease the probability of pages being swapped out, we chose the Windows kernel process, `ntoskrnl.exe`, and the user process `services.exe`. Figure 7 illustrates the difference in kernel and userspace translations. The difference between the two are due to the additional traversal of page entries as userspace will naturally be found further in memory. The important difference within this graph is, once more, the difference from Dom0 and the FVM whereby the the FVM architecture has a better



(a) Process Table (b) Module List

Fig. 6. Kernel structure traversal.



(a) Kernelspace (b) Userspace

Fig. 7. Time to read 100 bytes of a process.

throughput when translating virtual addresses than that of Dom0. It is not necessary to perform a page translation byte by byte, since there is only a need if the edge of a physical page is reached, but since caching was not involved in our design, it fits the purposes of measurement.

VIII. Discussion and Related Work

This research builds on Payne et al. [26]. We introduce specialised security domains that focus on the detection of symptoms and mobility schemes. Furthermore, the FVMs do not have a big a problem for waiting to be scheduled such as an application would on Dom0. It is hopeful that through better research into system symptoms, there will be a greater chance of detecting zero-day malware.

Hypervisor Security: The TCB (Trusted Computing Base) consists of the Xen hypervisor and Dom0. We have managed to add security to guests of the Xen hypervisor without dramatically increasing the size of the TCB. A couple of modifications to the Xen source code were needed in order to allow FVMs to introspect the memory pages of more than one target domain. We aim to achieve this same privilege without any modifications in future Xen releases. The only addition to the TCB were message receivers running in Dom0 for collecting symptom messages. Whilst we did not manage to remove complete reliance on Dom0 in our current implementation, there is no reason why these message receivers cannot run in a disaggregated domain as well. We intend to have a dedicated FVM message receiver running outside of Dom0 in our future work. Although the Mini-OS FVMs are privileged to inspect memory of guest VMs, they do not have additional privileges

which can be exploited. If the FVM security software can be exploited, the underlying hypervisor remains sufficiently isolated from attack.

Attacks against VMI: It is possible for malware to alter data structures and their respective pointers within an operating system. This technique has been shown to defeat virtual machine introspection techniques [27]. Although this technique can throw security monitors off course, this also requires significant patching effort from a malware writer. If VMs start off in a known good state and are constantly monitored by FVMs, then this image can be treated as a baseline. Subsequent changes in system runtime can then be modelled, and modifications to static data structures or large number of changes to various entry points can be treated as symptoms. This solution is likely to generate a lot of false positives and a solution will be needed to suitably model the rate of change within a typical image.

Xen Scheduler: We used the SMP credit scheduler for our tests. It is a work-conserving scheduler included in the Xen hypervisor. Further research on schedulers is required to better handle the case of FVMs. There has been previous work including co-scheduling that pairs a stubdomain with a domU VM [28], which could help in making sure that there is always at least one FVM per VM, whilst not encroaching on the scheduled CPU time the domUs have. Furthermore Cherkasova et al. provide a comparison of three schedulers in Xen [29].

IX. Conclusion

This paper presents a method of creating small virtual machines to inspect other guest virtual machines in order to discover symptoms of malicious behaviour. The proposed techniques involve a modularised architecture which makes use of Mini-OS on the Xen virtualisation platform. The architecture allows the reuse of code with the help of an API for conducting introspection. The lightweight footprint and simplicity of the Mini-OS Xen kernel cuts down the regular OS overhead substantially, making FVMs ideal Cloud security sensors. The infrastructure is evaluated and pros and cons of our design decisions are discussed.

References

[1] K. Harrison, B. Bordbar, S. T. T. Ali, C. I. Dalton, and A. Norman, "A Framework for Detecting Malware in Cloud by Identifying Symptoms," in *Enterprise Distributed Object Computing Conference (EDOC), IEEE 16th International*, 2012, pp. 164–172.

[2] Z. Wu, S. Gianvecchio, M. Xie, and H. Wang, "Mimimorphism: A new approach to binary code obfuscation," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 536–546.

[3] P. Szor, *The art of computer virus research and defense*. Addison-Wesley Professional, 2005.

[4] J. Baltazar, J. Costoya, and R. Flores, "The real face of koobface: The largest web 2.0 botnet explained," *Trend Micro Research*, vol. 5, no. 9, p. 10, 2009.

[5] N. Falliere and E. Chien, "Zeus: King of the Bots," Symantec Security Response, Tech. Rep., 2009, available at: www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/zeus_king_of_bots.pdf.

[6] P. Porras, H. Saidi, and V. Yegneswaran, "Conficker c analysis," SRI International, Tech. Rep., 2009, accessed: 23/03/13.

[7] Citrix Systems, "Xen Hypervisor," xen.org/products/xenhyp.html.

[8] Symantec, "Infostealer.Banker.C," www.symantec.com/security_response/writeup.jsp?docid=2010-020216-0135-99.

[9] Kaspersky, "Gauss: Abnormal distribution," Kaspersky Lab Global Research and Analysis Team, Tech. Rep., 2012, available at: www.securelist.com/en/downloads/vlpdfs/kaspersky-lab-gauss.pdf.

[10] Symantec, "Attacks on virtual machine emulators," www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf.

[11] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi, "A fistful of red-pills: how to automatically generate procedures to detect cpu emulators," in *Proceedings of the 3rd USENIX conference on Offensive technologies*, ser. WOOT'09. USENIX Association, 2009, pp. 2–2.

[12] 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*, ser. CCS '09. ACM, 2009, pp. 199–212.

[13] T. Garfinkel and M. Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection," in *In Proc. Network and Distributed Systems Security Symposium*, 2003, pp. 191–206.

[14] B. D. Payne, "Simplifying virtual machine introspection using libvmi," Sandia National Laboratories, Tech. Rep., 9 2012, available at: prod.sandia.gov/techlib/access-control.cgi/2012/127818.pdf.

[15] VolatileSystems, "The volatility framework: Volatile memory artifact extraction utility framework," www.volatilesystems.com/default/volatility.

[16] K. Kourai and S. Chiba, "Hyperspector: virtual distributed monitoring environments for secure intrusion detection," in *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, ser. VEE '05. ACM, 2005, pp. 197–207.

[17] VMware, "VMware VMsafe," www.vmware.com/technical-resources/security/vmsafe/faq.html.

[18] D. G. Murray, G. Milos, and S. Hand, "Improving Xen security through disaggregation," in *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 2008, pp. 151–160.

[19] S. Popuri, "A tour of the Mini-OS kernel," www.cs.uic.edu/~spopuri/minios.html.

[20] Z. Wang, "XenStore Socket," code.google.com/p/xenstore-socket/.

[21] B. D. Payne, "Virtual machine introspection tools," code.google.com/p/vmitools/.

[22] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Commun. ACM*, vol. 20, no. 10, pp. 762–772, Oct. 1977.

[23] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt, "Fast pattern matching in strings," *SIAM journal on computing*, vol. 6, no. 2, pp. 323–350, 1977.

[24] C. Vinschen and J. Johnston, "The newlib homepage," www.sourceware.org/newlib/.

[25] K. Kim, C. Kim, S.-I. Jung, H.-S. Shin, and J.-S. Kim, "Inter-domain socket communications supporting high performance and full binary compatibility on xen," in *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ser. VEE '08. ACM, 2008, pp. 11–20.

[26] B. D. Payne, M. de Carbone, and W. Lee, "Secure and flexible monitoring of virtual machines," in *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*. IEEE, 2007, pp. 385–397.

[27] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, D. Srinivasan, J. Rhee, and D. Xu, "Dksm: Subverting virtual machine introspection for fun and profit," in *Proceedings of the 2010 29th IEEE Symposium on Reliable Distributed Systems*, ser. SRDS '10, 2010, pp. 82–91.

[28] J. Nakajima and D. Stekloff, "Improving hvm domain isolation and performance," www.xen.org/files/summit_3/stub-xensummit-sept06.pdf.

[29] L. Cherkasova, D. Gupta, and A. Vahdat, "Comparison of the three cpu schedulers in xen," *SIGMETRICS Perform. Eval. Rev.*, vol. 35, no. 2, pp. 42–51, Sep. 2007.