VIRTUALIZATION BASED SECURE EXECUTION AND TESTING FRAMEWORK

Srujan Das Kotikela
Computer Science and Engineering
University of North Texas
Denton, TX, USA
srujankotikela@my.unt.edu

Satyajeet Nimgaonkar
Computer Science and Engineering
University of North Texas
Denton, TX, USA
satyajeetnimgaonkar@my.unt.edu

Mahadevan Gomathisankaran
Computer Science and Engineering
University of North Texas
Denton, TX, USA
mgomathi@unt.edu

ABSTRACT
Computer security aims at protecting confidentiality, integrity, and availability of sensitive information that are processed, used, or stored by computing systems. Computer scientists working in the field of computer security have successfully designed and developed software and hardware mechanisms to provide security in modern day computing devices. As compared to hardware security mechanisms, software-only security mechanisms are easy to implement and patch. But software-only security mechanisms cannot ensure protection against hardware-based attacks, thus rendering them vulnerable to such attacks. Hardware mechanism such as secure architectures aim to root the trust of the security solution in the hardware architecture. These security architectures typically deploy security mechanisms like encryption/decryption to protect confidentiality and hashing to protect data integrity. Though the security provided by hardware secure architectures is reliably high, they require modifications to the processor micro-architecture. Any changes to the micro-architecture is an extremely costly and time consuming process. Also, testing these hardware secure architectures is difficult as it requires testing the complete system including hardware, software and applications. Recently, virtualization has emerged to be an efficient and cost effective technology that allows emulating hardware mechanisms. It also enables emulating new hardware features in a virtualized environment. This makes the task of testing security architectures efficient and easy. In this paper, we use a virtualization software to build a Virtualization Based Secure Execution and Testing Framework for testing hardware secure architectures. Our framework provides a mechanism to plug-in secure architectures and monitor or test the system behavior by performing attacks on it.

KEY WORDS
Virtualization, secure architectures, hypervisor, virtual machine monitor.

1 Introduction
The current trends in computing have allowed computer systems to evolve into complex forms to provide ease of access and increased connectivity and availability to its users. Desktops, notebooks, netbooks, cell phones, PDAs, network sensors, network routers etc. have pervaded in our day-to-day lives. As the dependence on these computing systems increase, so also does the amount of sensitive information processed or stored by these systems. This information may also include confidential personal data like secret passwords, credit card numbers and bank account numbers etc. Thus computer security has now become ever more important [1] to protect these systems from confidentiality, integrity, and availability threats. Symantec’s “Internet Security Threat Report” [2] showcases that both threats and vulnerabilities are increasing rapidly. For example, in the year 2010, 286 million unique variants of malware has been detected, mobile devices vulnerabilities has increased by 42%, and a botnet with more than million computers was detected.

In general, computer security aims at protecting confidentiality, integrity, and availability of sensitive information processed by computing systems. Confidentiality is breached when information is accessible to an unwanted and unauthorized entity. This entity could be a human, a software program or another computing system. Similarly, integrity is infringed when information is modified by such an entity and availability is broken when this entity succeeds in making the computing system not serve legitimate user requests. Traditionally, researchers in the fields of computer security have tried software and hardware mechanisms for implementing security in computing systems. Software only security approaches typically deal with application level and Operating System (OS) level security mechanisms. Though, software-only mechanisms are easy to implement and patch, they suffer from heavy false-positives and false-negatives, thus making them vulnerable to software attacks and untrustworthy. The most common software attack that exploits the software vulnerabilities in application code and OS is a Buffer Overflow Attack [3].

Hardware security, on the contrary, includes Secure Processor Architectures such as ABYSS [4], AEGIS [5], Arc3D [6, 7], Hide [8] and XOM [9] assume only the processor chip as secure and all off-chip devices as untrustworthy. These architectures deploy hardware mechanisms like memory encryption to provide confidentiality and memory authentication [10] to ensure the integrity of the applications. Memory encryption encrypts the data before writ-
Proposed Framework

The proposed Virtualization Based Secure Execution and Testing Framework by modifying an open source Virtualization Software - Xen [15]. Our framework provides a generic interface to plug-in an existing secure architecture. Once plugged-in, the attack suite in our framework performs a series of attacks on a secure application/process running on the underlying secure architecture. A log of all these events is stored which can be later reviewed to judge if the secure architecture behavior is secure or not.

The rest of the paper is organized as follow. Section 2 describes our Proposed Framework. Section 3 presents the Implementation of our framework. Section 4 describes the Related Work followed by the Future Work in Section 5 and Conclusion in Section 6.

2 Proposed Framework

The proposed Virtualization Based Secure Execution and Testing Framework for testing hardware secure architectures is developed on top of Xen Hypervisor [15, 16], an open-source Virtualization Software. Therefore our framework uses some of the components already provided by Xen. Figure 1 below shows all the components of the proposed architecture along with their functioning. These components are Xen Hypervisor, MoCo VM, Application VM, Event Trigger Mechanism, Secure Architecture Plugin Interface (SAPI) and Attacker VM. The MoCo VM in turn consists the Monitor and the Controller while the Application VM consists the Secure Application running on top of the underlying Secure Architecture.

2.1 Xen Hypervisor/VMM

The Xen Hypervisor is the basic abstraction layer (virtualization layer) software that sits directly on the hardware below the operating system. It emulates the underlying hardware and is responsible for CPU scheduling and memory partitioning in order to allow multiple Virtual Machines to run on single hardware platform. Xen Hypervisor controls the execution of all the Virtual Machines running on it, however it has no knowledge of networking, external storage devices, video, or any other common I/O functions found in a computing system.

2.1.1 MoCo VM

A MoCo VM - Monitor/Controller Virtual Machine is essentially a specialized DOM 0 kernel found in the Xen architecture. DOM 0 also known as Domain 0, is a specialized Virtual Machine running on the Xen Hypervisor with special privileges to access physical I/O resources and communicate the other Virtual Machines running on the system. In its simplest form, a DOM 0 is a modified Linux kernel, that must be running on all Xen Virtualization environments before any other Virtual Machines could be started. In our framework, the DOM 0 kernel contains the Monitor and the Controller to form a MoCo VM.

2.1.2 Application VM

An Application VM is a DOM U kernel found in the Xen architecture. DOM U also known as the Domain U, is an unprivileged Virtual Machine running on Xen Hypervisor. Xen currently supports both para virtualization and full virtualization. In para virtualization, the guest kernel has to be modified in order to run on Xen e.g. Linux OS, Solaris, FreeBSD etc. Virtual Machines running such a kernel are known as DOM U PV Guest. Whereas in full virtualization, the guest kernel is not modified e.g. Windows, Virtual Machines running such a kernel is known as DOM U HVM Guest. In the proposed framework, a DOM U kernel is used to run the Secure Application that utilizes the security mechanisms provided by the underlying plugged-in secure architecture.

2.1.3 Event Trigger Mechanism

Event trigger provides a mechanism to initiate inter virtual machine communication. Traditionally Xen provides some mechanisms and tools like split driver, xenstore, grant table and ring buffers to carry out inter VM communication [17]. However the primary drawback of these mechanisms and tools is that they need the support from DOM U kernel to initiate communication. The proposed framework assumes...
all DOM U kernels to be untrustworthy and vulnerable to attacks. Hence we have avoided the use of these traditional mechanisms and have developed a new inter VM communication mechanism through the use of hypercalls and virtual interrupts (VIRQs), that are not dependent on the DOM U kernel. A hypercall works similar to a system call in a kernel or an OS. It is an interrupt typically INT 81h in Xen used to switch the control between the kernel and the Xen hypervisor. Hypercall interrupt the processor and are trapped in Xen using Hypercall Handler functions. VIRQ is a software interrupt notified by setting up a bit in the Virtual CPU data structure present in Xen.

2.1.4 Secure Architecture Plug-in Interface

The SAPI is a generic interface exposed by our framework to easily interface a secure architecture. SAPI is a collection of APIs which are used to modify virtual CPU (VCPU), virtual memory management unit (VMMU) and provide memory introspection functions into the modified components of the Xen. Secure architectures are typically changes in CPU and memory of the hardware which facilitate secure execution and isolation of a process. These components are available as software modules in Xen, it would be easy to modify them and program (or change) according to different secure architectures. As shown in Figure 1, the SAPI primarily has three components: modified VCPU API, modified VMMU API and modified memory introspection API. The modified VCPU API has all the set of functions which modifies the Virtual CPU provided by Xen. These modifications include providing encryption and decryption capabilities to the processor, add extra and secure registers, modify or implement new cache memories etc. The modified VMMU has set of functions which enhance the existing virtual memory management Unit provided by Xen. These changes include switching on/off the conventional virtual memory layout of the operating system, encrypting and decrypting functions for main memory and adding additional access restrictions to the memory access of secure process memory pages. Though there are memory introspection functions already available by the Xen VMM, these threats are invalidated once security architecture is invoked. Hence new set of newly customized/modified memory introspection functions are needed for each secure architecture.

2.1.5 Monitor

Monitor is a program running in the DOM 0 that receives notifications about the Secure Application through VIRQs. It is responsible for monitoring critical events pertaining to the Secure Application and detect any possible software attacks on it. It contains a monitoring script which is a collection of watch events and actions to be performed when a particular event has occurred.

2.1.6 Controller

The security architectures should be rigorously tested against various attacks. These attacks are almost similar for all the architectures. Hence we used a collection of such attacks called the Controller. The Controller is a collection of host and network based attacks launched from the DOM 0. While a secure application is running in the Applica-
Secure Application

Secure Application is a special process whose memory is required to be protected from attacks on the system. This process is aware of the security mechanisms provided by the secure architecture plugged in to our framework. Secure Applications or processes protect their confidential data by storing them in specialized protected memory regions.

Attacker VM

Similar to the Controller, the Attacker VM is used to launch inter VM attacks on the secure application running in the Application VM.

Framework Functioning

The first step of the functioning is to plug-in a secure architecture by using the generic interface provided by the SAPI. Once this is done, the VCPU, VMMU and virtual memory introspection functions are appropriately modified to align with the plugged-in secure architecture. All the information pertaining to the security mechanisms provided by the secure architecture is then reported to the Monitor through the Event Trigger mechanism. The Monitor thus now has complete visibility and understanding of the secure regions within the secure architecture. Now the Secure Application can start executing in the Application VM. Typically, the entire memory allocated to the Secure Application need not be confidential and hence has no need to be protected. A Secure Application identifies its critical confidential data and then transitions to secure execution state. In the proposed framework, this is achieved by invoking a secure hypercall `enter_vbase`. Once the Secure Application invokes this hypercall, the Secure Architecture and the Monitor are informed about the secure execution of the application. Its now the responsibility of the Secure Architecture to protect the confidential data of the application, while the Monitor is responsible for tracking all the necessary events related to the application. While the application is executing securely, the Monitor informs the Controller to conduct a series of attacks on the secure application. During this time, the Attacker VM is also initiated to carry out inter VM attacks on the secure application. The secure application can exit secure execution by invoking the `exit_vbase` hypercall. At this instant, the Monitor stops tracking the events while the Controller and Attacker VM stop the attacks on the application. Once the secure application terminates, all the events recorded by the Monitor are available for review in a log file. This file serves as a benchmark to judge the effectiveness of the security mechanisms provided by the secure architectures.

Attack Model

In practice Secure Applications are modified to directly communicate with the secure architectures. Ideally this communication should by pass the standard communication mechanisms used by other applications, i.e. through the operating system. The primary reason being these other applications and the OS may have vulnerabilities that could be exploited by an adversary and hence are untrustworthy.

Instead of rooting the trust of Secure Applications in the OS, it is rooted in the secure architectures plugged into the framework. The software components in a computing system are susceptible to attacks like spoofing, splicing and replay. In a spoofing attack, an adversary successfully masquerades as an authorized entity to intercept valid requests from another authorized entity and returns a faulty or a malicious response to that request. A splicing attack is the one in which an adversary intercepts valid requests from an authorized entity and returns a valid but an unwanted response to the request. And finally in a replay attack, an adversary returns stale copies of responses to the requests made by authorized entities.

Figure 2.3 shows the generic threat model considered for the proposed architecture. Here the Virtualization Layer, Xen VMM containing the secure architecture and the Secure Application are two trusted entities, whereas the OS i.e DOM U kernel and other applications are considered to be untrustworthy. The major attack sources are network attacks, spoofing attacks, splicing attacks and replay attacks. These attacks occur on Secure Applications and the untrusted entities in the system.

Implementation

We have implemented a secure architecture within the Xen VMM. This secure architecture provides two hypercall functions `enter_vbase` and `exit_vbase` to enter and exit secure execution. A hypercall handler is used to encode security architecture functionality. More robust and sophisticated security architectures can be done as separate modules (security components) with in the VMM. In any application a lot of code is derived from publicly available libraries. This code is not sensitive or critical from security
perspective. Hence, most of the code need not be secured. In the implemented example, we assume that secure process allocates a chunk of memory. This chunk of memory is used to store all the security critical data. This is very similar to the heap allocated and maintained by JVM. The framework protects this chunk of memory using the implemented secure architecture. For ease we have created the memory in the size of a page and aligned it to the beginning of the process execution.

The working of the secure process in conjunction with the security architecture is described in Algorithm 1 followed by code explanation. In the algorithm 1 a Secure Application identifies its critical confidential data and invokes a hypercall to report the memory regions to be protected. It sends the Virtual Addresses (VA) of this data to the DOM U Kernel that includes a specialized Kernel Module, which responds for translating this VA to a Physical Frame Number (PFN) or a Guest Frame Number (GFN). PFNs or GFNs are pseudo memory frames maintained by Xen to distribute real memory equally among its Virtual Machines. This PFN and VA is then sent to the Hypercall Handler in the Xen Hypervisor, where the PFN is in turn translated to real Machine Frame Numbers (MFN) corresponding to the Hardware Memory. These MFNs or hardware memory pages are marked READ-ONLY in order to protect them from being overwritten by attackers. Xen VMM provides a mechanism for making memory pages READ-ONLY in its pagetables. We have used this mechanism to simulate the effect of integrity verification through Hash Blocks. This helps in preserving the integrity of the data written to the memory. This way of implementing integrity verification helps us avoid the performance overhead induced by the Hash Block.

The memory pages are encrypted by the Encrypt/Decrypt block within the implemented secure architecture and hence even if an attacker reads them, would be unsuccessful in decoding sensible information out of it. At the same time, a VIRQ is sent from Xen to the DOM 0 Kernel. A VIRQ Handler present in DOM 0 receives this VIRQ and sends a SIGNAL to the Monitor to notify it about the Secure Applications protected memory running on the DOM U. The Monitor then invokes a hypercall to collect specific details about the protected memory region. These details like VA, MFN and memory status etc. are copied to its hypercall handler from the hypercall handler of the hypercall invoked by the Secure Application. Thus the Monitor now has all the information to start monitoring these protected memory regions of the Secure Applications and is responsible for detecting any possible software attacks that may occur on these protected memory regions. Though this architecture is built by modifying Xen Hypervisor, it only uses the generic mechanisms of Xen that all the Virtualization Softwares must provide. Hence it is possible to interface this architecture with other Virtualization Softwares, with minor changes. This provides an excellent and cost efficient platform for secure architecture vendors to test their architectures and detect any possible vulnerabilities that may exist in them.

**Algorithm 1 Secure Process Execution**

1. Secure Process Begins
2. Allocate Secure Memory
3. The virtual address of secure memory is passed through vbase.sys system call
4. Virtual address of secure memory is received in kernel module
5. Physical Frame Number (PFN) is computed for the received virtual address
6. HYPERVISOR_vbase_op hypercall is used to pass the PFN to the hypervisor
7. PFN is received in the hypercall handler
8. Machine Frame Number (MFN) is computed for the received PFN
9. MFN is made read-only
10. Virtual Interrupt (VIRQ) is sent to Dom0
11. VIRQ is received in the Dom0 kernel
12. Dom0 kernel sends signal to the Monitor
13. Monitor prints messages to user

### 3.1 Source Code Implementation

In the listing 1, a chunk of memory (secure_mem) of PAGE_SIZE is allocated and the beginning of the chunk of the memory is aligned to beginning of a page.

```
char secure_mem[PAGE_SIZE]
secure_memory __attribute__((
__aligned__(PAGE_SIZE)));
```

Listing 1. Secure Memory Allocation

In listing 2 we declared a integer pointer (secret_key) and type casted the char buffer pointer to an integer pointer (secret_mem). The secret_key is assigned the address of the secure memory. Now the contents of the address contained in secret_key will be stored in the chunk of memory allocated in listing 1.

```
int * secret_key = (int *) secret_mem;
*secret_key = 654321;
```

Listing 2. Secret Key Linked to Secure Memory

The address of the secret key is to be protected. We have to make it read only. In listing 3, we pass the secret key to the kernel module using the system call, sys_vbase. sys_vbase is custom system call which takes virtual address as a parameter. If the secret_key is protected, the secure process should print the same values before and after the system calls.
printf("The secret_key before system call is: %d", *secret_key)
//prints the value 654321
sys_vbase(secret_key);
*secret_key = 123456;
printf("The secret_key after system call is: %d", *secret_key)
//prints the value 654321
//In xen
send_guest_global_virq(dom0, VIRQ_VBASE);
//In Dom0 kernel
bind_virq_to_irqhandler(VIRQ_VBASE, 0, vbase_handler, NULL, NULL, 0);

Listing 3. Decode Address of Secret Key

After the system call, the control is passed to the kernel (into the system call handler). In the kernel, the Physical Frame Number (PFN) or the frame number in the RAM of the DomU for the virtual address is computed. PFN is also called as Guest Frame Number (GFN). Computed PFN is passed down to the VMM using the hypercall. Ideally we do not want to invoke hypercall from the kernel. However, Xen allows hypercall to be made only from the privileged level i.e. from the kernel.

Listing 4. Translate VA to PFN

Hypercalls are trapped into the VMM, thus giving VMM control. The blocks of VMM where hypercalls are directed to is called as hypercall handler. The PFN is received in the hypercall handler and Machine Frame Number (MFN) is computed which is the hardware machine’s RAM frame number. This MFN is marked read-only by the xen’s in built function. This function requires previous type (read-only, writeable) of the MFN. Hence we first invoke a function which returns the old type of the MFN. The MFN is passed along with the old type and new type (p2m_ram_ro). This will update the MFN type in xen’s tables. Any further requests to overwrite the pages identified by the MFN are dropped by xen. Attempts for trying to write to the read-only MFN are reported to Dom0 through xend.

Listing 5. Translate PFN to MFN

struct vcpu *v = current;
struct domain *d = v->domain;
p2m_type_t old_type;
mfn_t mfn;
mfn = gfn_to_mfn(d, gfn, &old_type);
p2m_change_type(d, gfn, ot, p2m_ram_ro);

Listing 6. Dispatch VIRQ from XEN

In listing 6 the code sends a global VIRQ to the Dom0. In xen VIRQ’s are handled by the guests as interrupts. So we need to bind the VIRQ to an IRQ Handler. After the VIRQ is populated from the hypercall handler, the Dom0 is resumed and an interrupt (dynamically bound by the kernel) is issued. This will trap the control into the IRQ handler bound earlier.

Listing 7. Dispatch a SIGNAL to the Monitor

In listing 7 the task_structure of the Monitor process is found using find_task_by_pid function provided by the kernel (note that pid of the monitor can be sent through a system call or by writing to kernel filesystems). After finding the task_structure a signal is issued to monitor process. This will notify that the virtual address of the secure process has been made read-only.

4 Related Work

4.1 Process-level Isolation using Virtualization

dfork [18] is a clone of posix fork in which a separate kernel and filesystem is allocated to every new process to isolate its operations and interactions of other processes with the secured process. This method also gives an ability to review the changes done by the application before committing to the underlying hardware. Thus giving us the control to accept some changes, keep some isolated, and discard others entirely. dfork is also recursive, in the sense that a secured process can in turn spawn another secure process. Our architecture doesn’t try to modify or override existing OS functionalities. It adds minimum overhead (hypercalls and VIRQs) to secure applications.

4.2 Quebes OS

Joanna Rutkowska introduces a new (modified Linux) operating system called Quebes OS [19]. It is very similar to dfork in a way that every process that is being secured gets its own kernel and environment in a virtualized container.
The quebes approach uses a lightweight templating mechanism to create new virtual machine for every process to be secured on the fly. A copy of light-weight operating system (assumed to be secure) is saved in the control of VMM and whenever a user wants to run an application securely, a new virtual machine based on the secure copy is instantiated and the application is run in the virtual machine and only that application is run in that virtual machine. The (isolation) security provided by the Quebes OS is based on the (widely accepted) fact of strong isolation provided by Xen. Our architecture doesn’t use any more resources than actually necessary for the application. Instead of running the application in its own virtual machine to isolate, our architecture focuses on security critical code and tries to isolate only security sensitive code in the VMM using existing VMM functionality.

4.3 Singularity

Microsoft OS research proposes a novel approach to solve the process isolation problem by introducing a new concept of Software Assisted process isolation as opposed to widely accepted, de-facto, standard of hardware assisted process isolation. Singularity [20] addresses the performance penalties of hardware based isolation and how this penalties force OS developers to break the intended design and make the OS insecure. Software assisted isolation tries to exploit the advancements in language technologies such as objects and messages and advocates usage of advanced languages to build secure operating systems. In our architecture we don’t have to introduce new operating system, environment or applications. We have shown that applications can be secured with in the existing environments without much effort.

4.4 Framework for Design Validation of Security Architectures

Framework for Design Validation of Security Architectures [21] showed how security architectures can be implemented in virtualized environments. The paper explained how security components can be embedded in virtualization layer to mimic hardware security architectures in the virtualized environments. The framework is then shown to be used to test security architectures and rapid prototyping of security architectures. This was implemented using VMware and is more of a testing framework.

5 Future Work

Security through virtualization technology is a very promising and upcoming field. There is lot of research going on exploring many ways to leverage the widely used virtualization. It would be no surprise if future computer systems ship with virtualization software along with firmware. Which means virtualized solutions for security would be more common place in coming future. We can build more functional security components in the virtualization layer. These components can be exported to applications through an API. This would help application developers in building secure applications with minimum modifications.

The monitor can be made more powerful by adding more functionality like looking into the entire process memory and detect malicious activity while safe guarding the security intensive memory regions. Monitor can also be made interactive, where it receives a set of commands to monitor specific functions/memory regions, events through an input file. We can create an environment which automates various types of attacks on the secure process and let the monitor get the effects of the attack. This gives quantified data about the effectiveness of the security architecture. As many businesses are moving towards cloud computing [22] which is majorly deployed over virtualization layers, we can implement our architecture in public and private clouds to ensure various challenges such as: confidentiality and data integrity [23]. Also, in future, when virtualization becomes prevalent on the desktop systems our architecture can be used to ensure the security of the applications.

6 Conclusion

We have successfully shown how security architectures can be built and implemented easily and efficiently using virtualization technology. We have proposed a Virtualization Based Secure Execution and Testing Framework and implemented its prototype using xen virtualization layer. The framework proves to be secure against unauthorized overwriting of security sensitive memory locations. Since the application is secured in the virtualization layer, it is resistant to OS level attacks also. This also removes operating system out of trusted computing base, which proved to be a big challenge for years. With this we conclude that we can provide security to applications by leveraging virtualization technology. Also our framework does not impose additional performance overheads on the system as compared to the traditional virtualized computing systems. Hence this framework can be successfully adopted to test secure architectures without imposing performance overheads on the system.

References


