Fine-Grained I/O Access Control of the Mobile Devices Based on the Xen Architecture

Sung-Min Lee, Sang-Bum Suh, Bokdeuk Jeong, Sangdok Mo, Brian Myungjune Jung, Jung-Hyun Yoo, Jae-Min Ryu, and Dong-Hyuk Lee
Samsung Advanced Institute of Technology (SAIT)
Samsung Electronics
Gyeonggi-Do, Korea
{sung.min.lee, sbuk.suh, bd.jeong, sd.mo, brian.m.jung, yjhyun.yoo, jm77.ryu, dh5050.lee}@samsung.com

ABSTRACT

System virtualization is now available for mobile devices allowing for many advantages. Two of the major benefits from virtualization are system fault isolation and security. The isolated driver domain (IDD) model, a widely adopted architecture, enables strong system fault isolation by limiting the impact of driver faults to the driver domain itself. However, excessive I/O requests from a malicious domain to an IDD can cause CPU overuse of the IDD and performance degradation of applications in the IDD and other domains that share the same I/O device with the malicious domain. If the IDD model is applied to mobile devices, this failure of performance isolation could also lead to battery drain, and thus it introduces a new severe threat to mobile devices. In order to solve this problem, we propose a fine-grained I/O access control mechanism in an IDD. Requests from guest domains are managed by an accounting module in terms of CPU usage, with the calculation of estimated CPU consumption using regression equations. The requests are scheduled by an I/O access control enforcer according to security policies. As a result, our mechanism provides precise control on the CPU usage of a guest domain due to I/O device access, and prevents malicious guest domains from CPU overuse, performance degradation, and battery drain. We have implemented a prototype of our approach considering both network and storage devices with a real smart phone (SGH-i780) that runs two para-virtualized Linux kernels on top of Secure Xen on ARM. The evaluation shows our approach effectively protects a smart phone against excessive I/O attacks and guarantees availability.

Categories and Subject Descriptors

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

General Terms

Security, Design, Experimentation, Reliability

Keywords

Virtual machine, virtual machine monitor, I/O access control, smart phone security

1. INTRODUCTION

System virtualization has become a compelling technology, quickly pervading many kinds of devices with the promise of cost savings, security, and other benefits. Owing to these advantages, even consumer electronics devices like mobile phones have come to adopt the technology. Currently, Xen [1], a popular open source *virtual machine monitor* (VMM), has been ported to the ARM architecture used for secure mobile phones [2,3,4,5].

There have been many contributions to building secure computing environments with the virtualization technology, which inherently provides two security primitives: isolation and outside observance. Secure Xen on the ARM platform [2,3,4,5,6] supports enhanced security features for mobile devices with mandatory access control through five formal access control models [6] and secure boot. It guarantees confidentiality by confining resource sharing among domains with the access control on physical/virtual devices. It also supports the integrity of a VMM including access control modules as it detects any alternation of the VMM during bootstrap time.

However, there have been fewer efforts in virtualization security for mobile devices that guarantee availability against Denial of Service (DoS) attacks from malware considering that mobile devices have limited system resources. The use of techniques similar to an *isolated driver domain* (IDD) in Xen 3.0 appears to be a growing trend with many cited advantages including portability, reliability, and extensibility [3,7,8]. The IDD has a backend driver that multiplexes I/O for multiple frontend drivers in guest domains over a real device driver. Despite the benefits of IDD, it allows severe attacks on resource-limited mobile devices because of its structural problem with client/server-based I/O processing and the complex CPU usage model. If a compromised guest domain keeps sending intensive I/O requests through its frontend device driver, IDD has to access the real I/O device with its native device driver on behalf of the guest domain to process the I/O requests. This means malware installed in the guest domain can cause the IDD to excessively consume system resources, including CPU and I/O resources. Furthermore, excessive use of an I/O device by the malicious domain can also degrade performance in domains that share the same I/O device with the malicious domain. As a result, a mobile device cannot support important applications or services in a guest domain in a timely manner, and the attack may drain the battery in a short time. These denial-of-service attacks and the failure of performance isolation prevent guaranteed availability of a mobile device, even if a VMM provides fault isolation and mandatory access control.

This paper makes three main contributions. First, we perform I/O intensive attacks targeting a mobile device based on the Xen architecture. We find that excessive I/O requests from malware installed in a guest domain deteriorate performance of the entire mobile device and drain battery as well, despite Xen's fair allocation of system resources such as memory and CPU to each domain. We demonstrate the amount of CPU overuse, performance degradation, and power consumption that can occur under excessive I/O attacks on storage and network devices. Second, we propose a fine-grained I/O access control mechanism in an IDD that enables mobile devices to provide good performance of important services and applications, and to preserve durable battery life even under malware attacks from a compromised guest domain. By investigating the correlation between CPU usage and I/O requests from each guest domain, our mechanism provides accurate access control enforcement of I/O requests. Third, we perform experiments on CPU usage, performance, and power in three scenarios: normal, under attack, and access controlled. Our analysis shows our mechanism precisely controls CPU usage, and it provides good performance and power preservation under our access control enforcement, guaranteeing resources are available for critical services and applications.

We have implemented our fine-grained I/O access control module on a real smart phone—a Samsung SGH-i780—that runs two para-virtualized Linux kernels on top of Secure Xen on ARM. The evaluation shows our approach effectively protects a smart phone against a new kind of DoS attack that sends excessive I/O requests to drain limited system resources in virtualization environment. Consequently, it guarantees good performance and availability in most any circumstances.

The rest of this paper is organized as follows. We describe the limitations of the existing Xen I/O architecture and a threat model in Section 2. We present the design and implementation of our fine-grained I/O access control mechanism for a mobile device based on system virtualization in Section 3. We evaluate the effectiveness and performance of our approach in Section 4. We discuss advantages and disadvantages of our approach in Section 5. We present related work in Section 6, and our conclusions in Section 7.

2. MOTIVATION

In this section, we first review the Xen I/O architecture and its security defects. We then describe a threat model for a mobile device.

2.1 Xen I/O Architecture and Its Problem

The I/O model in Xen has evolved to guarantee safety of a device. Originally, Xen 1.0 contained device driver code and provided shared device access. However, device drivers are one of the most troublesome aspects of commodity operating system — Chou et al. [9] estimated that more than 85% of the bugs in an operating system are driver bugs. As device drivers reside in a VMM running with the VMM privileges, bugs in drivers can easily cause system crashes or panics [10,11]. In order to reduce the risk of device driver failure/misbehavior and to address problems of dependability, maintainability, and manageability of I/O devices, Xen moved to a new architecture employing isolated driver domains (IDDs) that limit the impact of driver faults by introducing a frontend/backend device driver model [7,8]. The IDD has a backend driver that multiplexes I/O for multiple frontend drivers in guest domains over real device driver. Due to these advantages, Xen and other virtualization technologies [12,13] leverage the IDD model.

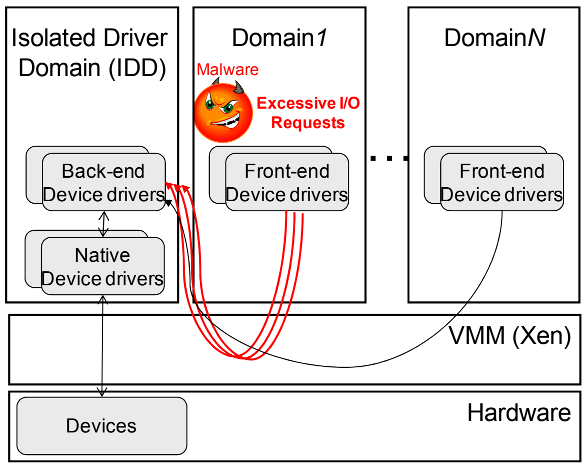

Figure 1. Xen I/O architecture and malware attack.

Unfortunately, this I/O model results in severe problems when malware in a guest domain tries to abuse system resources by sending an extreme number of I/O requests to the IDD. This problem is most significant in the case of mobile device environments. The result of a malware attack is exacerbated when an IDD serves as a secure domain, which contains important mobile phone services such as voice call and secure transaction, or as a control domain like proposed in [2,6]. The problems are as follows:

1. CPU overuse: A guest domain compromised by an attacker can take a greater share of CPU time than initially allocated. Consider when the guest domain is limited to 50% of the CPU. If the guest domain fully uses its CPU time and IDD consumes 30% of CPU to process the I/O request from the guest domain, the overall CPU consumption of the guest domain can take up to 80% total.
2. Performance degradation: The performance of other domains that share the same I/O device with the malicious domain would be also affected as well as IDD. Furthermore, if a secure domain shares an I/O device with the malicious domain, the security and availability of very important services such as voice call cannot be guaranteed.
3. Battery drain: The above attacks can be devastating to resource-limited mobile devices by exhausting energy.

2.2 Threat Model

We assume that unauthorized software downloaded from the Internet (e.g., via Google Android Market, Apple AppStore, etc) has the potential to be malicious [14]. Once, it is installed in a guest domain on a mobile device, the software (malware) not only can have complete control of the guest domain but also tries to attack other domains including control domain and secure domain.

Table 1 summarizes the potential threats to mobile devices based on system virtualization technology. Although all of these threats are hazardous to the mobile devices, we focus only on the Threat 4 in this paper since existing security approaches have mitigated Threat 1-3.

Specifically, Threat 1 can be solved by using access control mechanisms like [2,6,15] (both in the VMM and in the kernel) and end-to-end encryption mechanisms. In particular, [6] protects a mobile device against this threat by running two isolated domains, called Secure Domain and Normal Domain, on top of the VMM. The access control module inside the VMM forbids malware in the Normal Domain from accessing the resources that the Secure Domain uses (e.g., providing keystroke logger protection). Threat 2 can be thwarted by adopting integrity measurements such as [2,16,17]. To mitigate the Threat 3, device drivers can be exported from a VMM [2,10,11]. Hence, these threats are not covered in this paper.

On the other hand, there are no appropriate mechanisms for preventing denial-of-service (DoS) attacks on a mobile device. In particular, there has been no effort to protect a mobile device against CPU overuse, performance degradation, and DoS attacks caused by excessive I/O requests due to the misuse of an IDD architecture by malware.

One example of this kind of attack is the *Cabir* worm [18], which seriously reduces the battery life of mobile phones when it broadcasts messages through the Bluetooth channel at frequent intervals [19]. This threat is very severe since it could devastate resource-limited mobile devices. In the case of a severe version of this attack, a user cannot use the important services of a mobile phone, such as voice calls, security, and so on. Thus, we aim to ensure appropriate performance and availability of a mobile device even under malware attacks by providing fine-grained I/O access control at an IDD.

Table 1. Threat analysis

	Threats	Impact
1.	Sensitive information disclosure by malware (e.g., financial information)	High
2.	Alternation of core system modules by malware	High
3.	Bugs in device drivers or an operating system (Fault propagation)	Medium
4.	Performance degradation and denial of service by malware	High

3. PROPOSED ARCHITECTURE

In this section, we present overall architecture of our approach. We then explain in detail how we designed and implemented the fine-grained I/O access control mechanism.

3.1 Overall Architecture

Overall architecture of our approach is shown in Figure 2. Our I/O access control module consists of 5 major components: 4 components (Policy Manager, I/O Requests Measurement, I/O Accounting, and Access Control Enforcer) in the IDD and 1 component (I/O Requests Generator) in a guest domain.

In sum, we assume that the formulas (regression equations) to calculate the CPU time needed to process a request for an I/O device are already installed in a mobile device. However, a new I/O device may be added to the mobile device, or a user may request an update to the formula for certain reasons. In that case, the I/O requests generator in the guest domain and the I/O requests measurement module in the IDD are used. If the I/O requests generator sends pre-defined requests of various sizes to each I/O device, the I/O requests measurement module in the IDD generates regression equations to estimate CPU time per each I/O request size.

Using the regression equations, the I/O accounting module estimates how much CPU time is needed to process requests, which are then sent to a back-end device from a front-end device. It then asks the access control enforcer to decide whether the requests should be granted or delayed according to an I/O access control policy. The access control enforcer performs I/O scheduling for each device driver considering the estimated CPU time by the accounting module and the access control policy.

The policy manager is responsible for updating the I/O access control policy. The update or change of policy can be done by a device owner, certified operator, or manufacturer through a secure channel.

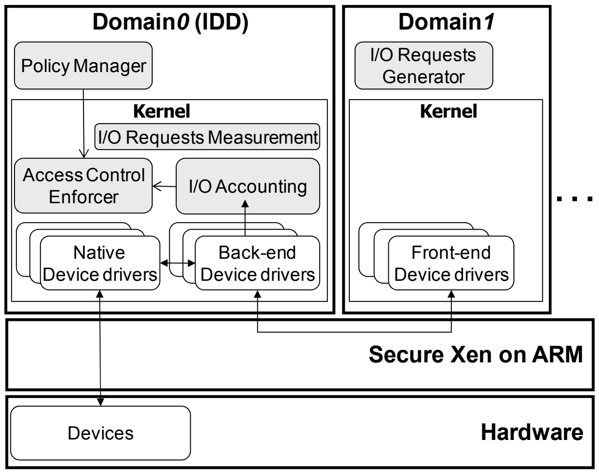

Figure 2. Overall architecture.

3.2 Regression Analysis

Acquiring information on CPU consumption by each virtual I/O operation is prerequisite to control virtual I/O usage of each guest domain with respect to the CPU consumption in Domain0. In this subsection, we describe the methodology of the CPU usage measurement and its application to a network device and a storage device.

3.2.1 Estimation of CPU Consumption

To obtain the CPU usage, we estimate CPU consumption from results measured ahead of time, rather than performing measurements at runtime.

Although accurate measurement is essential to virtual I/O access control, runtime measurement at every I/O operation causes serious CPU overhead. Even if we take simple timestamps between the start of a virtual I/O operation and the end of it, it takes an additional 16 usecs on the virtualized SGH-i780 with Secure Xen on ARM, and the measurement invocation mechanism would require additional CPU time. Moreover, frequent I/O requests exacerbate this situation. On the other hand, our estimation approach can considerably reduce those runtime measurement overheads by doing simple calculations. The normalized standard distribution of the measured result from our experiment showed that 90% percent of requests clustered by device type and its operation type deviates from their average less than 10%. Considering the above performance efficiency and the tendency of pre-measured data to be fairly close to the average, we expect that estimation at the expense of inaccuracy is acceptable.

To compensate for the discrepancy between the estimation and the runtime measurement, we suggest that I/O access control module takes a sufficiently long enough period of virtual I/O scheduling, so that the accumulated CPU time of estimation on I/O operations could be asymptotic to the sum of runtime measurement of each I/O operation.

3.2.2 Regression Analysis

The required CPU time for an I/O operation can vary with the operation type and the request size of each operation type on an I/O device [20]. The CPU consumption of I/O requests clustered by operation types of a virtual device can be expressed as a function of request sizes. Also, the function could be achieved through regression analysis from the measured data that is acquired by giving various size requests per operation type of a virtual I/O device.

We first performed measurements by putting timestamps on the critical path of virtual I/O operations, and then screened data that is not greater than a 10% deviation from the average. Those data greater than a 10% deviation usually resulted from a domain schedule timeout in Secure Xen on ARM, which causes excessively long periods of measurement, or dropping requests due to the limited capacity. After gathering the measured data, we performed regression analysis through MINITAB while choosing regression model of which R-squared is no less than 90%.

The measurement environment is as follows: two Linux virtual machines (version 2.6.21) running on Secure Xen on ARM, which is implemented on an SGH-i780 smart phone. Linux instrumentation, Kprobes [21], is used for timestamping on critical paths. As wireless network devices and storage devices have long been the target of resource-drain attacks such as sleep-deprivation attack over wireless network devices or disk space attack, we chose these two devices as the subjects of I/O access control.

3.2.3 Regression Analysis Result: Network

Virtual network devices for guest domains, each of which is shown as a virtual network interface in Domain0, are configured to access outside network through IP forwarding and Network Address Translation (NAT) via iptables settings in Domain0. Because we chose to control virtual I/O access at backend drivers, we only monitor outbound packet transmission; otherwise controlling inbound packets at backend would be too late to prevent CPU consumption at Domain0.

We first analyzed function call path from the network backend to the wireless network device transmission to place measurement instrumentation and calculate virtual network CPU consumption. Figure 3 describes the major call path for Tx packets of guest domains.

Once the backend device driver receives an outbound packet from a guest domain, it is put into Rx socket buffer (skb). The backend driver registered itself as a virtual network interface in Domain0 and the Tx packet of a guest domain is received by the virtual network interface. The network packet in the Rx socket buffer queue is then handled by *net_rx_action()* at soft IRQ routine, and is forwarded to output queue to be transmitted outside by iptables setting. Thus we divide the network call path into two phases; from the backend *netif_do_netdev()* until putting into the Rx socket buffer, and from *net_rx_action()* to outer packet transmission.

To obtain the result of the regression formula, we sent out UDP packets of various sizes (128, 256, 512, 1024, and 1472 bytes) with a user-level networking application (i.e., I/O Requests Generator in Figure 2).

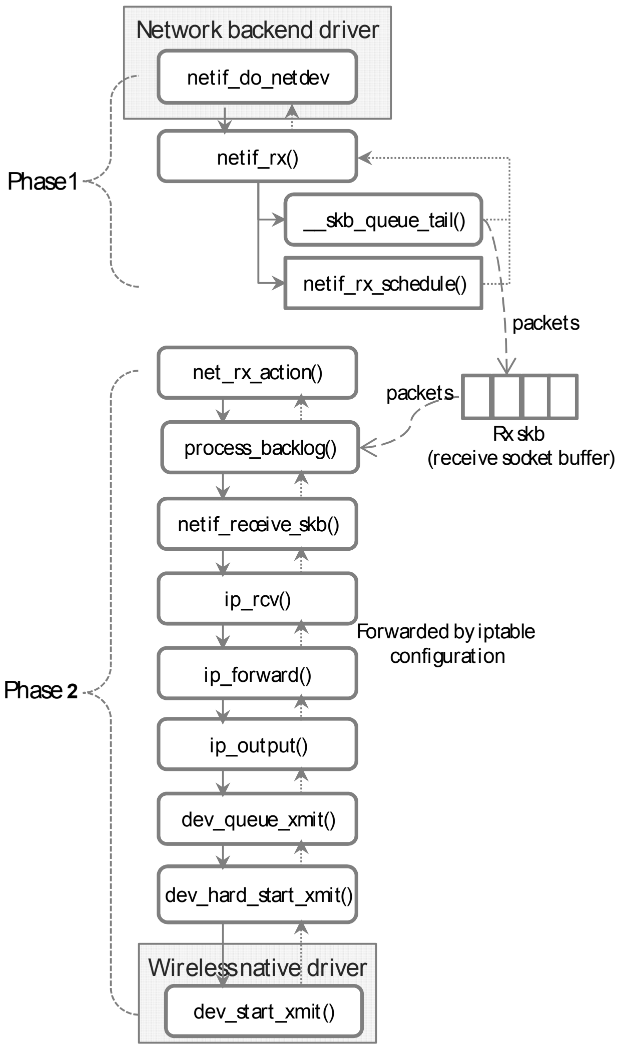

Figure 3. Function call path of outbound virtual network packets.

Because communication between the virtual network backend and the frontend is done at network link layer and the maximum size of network frame is 1500 bytes, we did not need to perform experiment with sizes beyond 1500 bytes, which would only be fragmented into packets less than 1500 bytes each in guest domains. We separately put timestamps at the start of *netif_do_netdev()* and on its return, and at the start of *net_rx_action()* and on its return using KRET in the Linux Kprobes instrumentation. The difference of the timestamps in each phase is then summed. The result is shown below:

$$f_{NET,TX}(bytes) = 303.18 + 0.01*bytes \quad (1)$$

3.2.4 Regression Analysis Result: MTD

The SGH-i780 uses NAND flash for its storage. We performed measurement at the Memory Technology Device (MTD) layer whereas Secure Xen on ARM virtualizes NAND flash at the MTD layer. The guest domain (Domain1) can access the 9th partition of NAND, which is seen as mtd0 by Domain1 and mtd8 by the control domain, Domain0. So, when Domain1 sends an MTD operation, a partition operation corresponding to the MTD operation in Domain0 is first invoked by the MTD backend driver, then a function in NAND subsystem, and finally a function in NAND native device driver.

MTD has 10 operation types for NAND: READ, WRITE, ERASE, BLOCK_ISBAD, BLOCK_MARKBAD, READOOB, WRITEOOB, SUSPEND, RESUME, and SYNC. The seven most frequent function calls except SUSPEND, RESUME, and SYNC are the subjects of our measurement. The call paths of MTD_READ function is depicted in Figure 4.

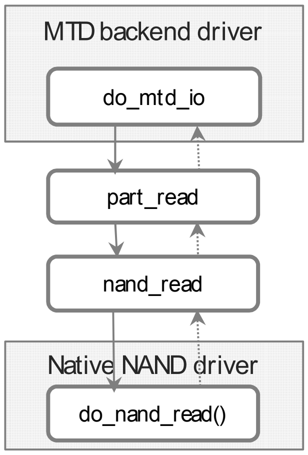

Figure 4. Function call path of MTD_READ.

The READ operation in Figure 4 is converted into the partition operation and then into the NAND operation, and other MTD operations from a guest domain are also converted into partition operations and NAND operations. Thus, we measured CPU time from the start of *do_mtd_io()* until the return of the function using KRET in Linux Kprobes instrumentation.

We chose the request sizes of each operation type after observing typical MTD requests: 512, 1024, 2048, 3072, and 4096 bytes for READ, 2048, 4096, 6144 bytes for WRITE; and 12, 24, 64 bytes for READOOB and WRITEOOB operations. The regression analysis from the measured data is shown below:

$$\begin{aligned}
&f_{MTD,READ}(\text{bytes}) = 116 + 0.24*\text{bytes}\\
&f_{MTD,READOOB}(\text{bytes}) = 533 + 0.06*\text{bytes}\\
&f_{MTD,WRITE}(\text{bytes}) = 160 + 0.24*\text{bytes}\\
&f_{MTD,WRITEOOB}(\text{bytes}) = 585 \qquad (2)\\
&f_{MTD,ERASE}(\text{bytes}) = 153.33 + 0.02*\text{bytes}\\
&f_{MTD,ISBADBLOCK} = 58\\
&f_{MTD,MARKBAD} = 60
\end{aligned}$$

The function WRITEOOB turned out to be independent from the input variable *bytes*. Given that the request size of OOB operations is relatively small, we can conjecture that the request size to WRITEOOB operation has less impact on the CPU consumption than WRITE operation. Because BLOCK_ISBAD and BLOCK_MARKBAD are checked in the unit of a block, the regression expressions are constant.

3.3 I/O Requests Accounting

We assume that each I/O operation is atomic and thus does not affect subsequent I/O operations. Therefore, the total CPU usage on Domain0 due to a virtual I/O device access by a guest domain becomes sum of the CPU usage of individual requests while the CPU time of an individual request is calculated from an independent variable *request size.*

$$vIO(i) = \sum_{j=1}^{n} f_{OP_j}(x_{OP_j}) \qquad (3)$$

$$\begin{cases}
vIO(i)\text{: total CPU consumption due to a virtual I/O device access by domain } i\\
n\text{: the number of requests to a virtual I/O device from domain } i\\
OP_j\text{: operation type of the } j^{th} \text{ request}\\
f_{OPj}\text{: function of CPU consumption regarding to the operation type}\\
x\text{: request size of an I/O operation}
\end{cases}$$

The function *f* is achieved through the regression analysis in Section 3.2.

3.4 I/O Access Control Enforcement

We use the regression formulas introduced in Section 3.3 for estimating CPU consumption to control virtual I/O access. In this subsection, we describe how the virtual I/O access control works. This subsection introduces an I/O access control framework, and then explains its implementation detail.

3.4.1 I/O Access Control Framework

Figure 5 depicts relationship among I/O access control module (I/O ACM) components, and between the I/O ACM components and virtual device drivers. I/O ACM is composed of a PolicyLoader, a ResourceAccount, and a virtual I/O scheduler. The PolicyLoader is responsible for loading policies from storage, the ResourceAccount for bookkeeping virtual I/O usages of each guest domain, and the virtual I/O scheduler for admission control on access to virtual I/O devices by guest domains.

The sequence for component invocation is as follows:

1. When the control domain (i.e., Domain0) boots up, the PolicyLoader accesses the SecureStorage, provided by Secure Xen on ARM, to retrieve I/O access control policies and to load them onto memory.
2. A virtual device driver registers itself on the list of controlled resources during its backend initialization; meanwhile ResourceAccount retrieves regression formulas for the virtual I/O device, which are used for CPU usage estimation during runtime.
3. If a guest domain's frontend driver connects to the backend driver, the virtual I/O scheduler fetches policies about the guest domain's virtual device access from PolicyLoader.
4. Whenever the backend driver receives I/O requests from a guest domain, it invokes the virtual I/O scheduler. Based on the policy and the guest domain's current usage of a virtual device, the virtual I/O scheduler determines if the request should be handled immediately or delayed for a while.
5. After the backend driver finishes handling the I/O request from the guest domain, it accounts guest domain's virtual device usage in terms of CPU time via ResourceAccount and updates scheduler credit. Note that in our current implementation, the ResourceAccount is not utilized at this stage, and is designed as future extension.

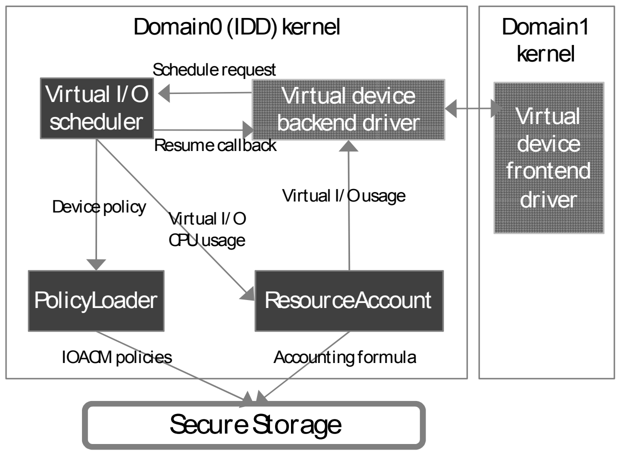

Figure 5. I/O ACM components relationship.

3.4.2 I/O ACM Implementation

The I/O ACM residing in the Domain0 Linux kernel runs on SGH-i780 smart phone, whose system architecture is para-virtualized with Secure Xen on ARM. The subjects of access-controlled resources are a virtual network device and a virtual MTD device. Implementation details of components in I/O ACM are below.

3.4.2.1 PolicyLoader

As the VMM manages a secure storage in Secure Xen on ARM architecture, PolicyLoader requests the VMM for I/O ACM policy stored in the secure storage, and loads the binary I/O ACM policy on Domain0 memory. When there is a notification of policy update from the VMM, PolicyLoader reloads policy from the VMM and notifies the virtual I/O scheduler to replace the scheduling policy.

3.4.2.2 Access Control Invocation

To invoke I/O ACM, we injected I/O ACM functions, such as *register_ioacm()* for registering a device as an access controlled resource, *check_permission()* for acquiring permission to process requests, and *account_resource()* for recording the resource usage. When a backend device starts to run, the *register_ioacm()* function retrieves the accounting information of a virtual device from ResourceAccount. On the initiation of inter-domain connection between a frontend device driver and the backend driver, the backend driver calls the *register_frontend()* function to notify virtual I/O scheduler of a new subject of I/O access control and to submit the scheduling policy. The *check_permission()* function is called before processing I/O requests and the *account_resource()* function is invoked after finishing the I/O requests.

3.4.2.3 Virtual I/O Scheduler

The virtual I/O scheduler decides the execution time of an I/O request based on I/O access control policy. It is currently implemented as a simple credit scheduler. Pseudo scheduling algorithm is described in Figure 6.

I/O ACM scheduler first calculates required CPU time to operate an I/O request and then convert the time into a credit score. After that, it compares the required credit with the currently available credit.

Scheduling:

```
CPUTime := calculate_required_CPUTime(virtual I/O request)
required_credit := convert_time_to_credit(CPUTime)
if (start_of_new_scheduling_period(current_time, I/O device, guest domain))
        available_credit := credit_by_control_mode(I/O device, current policy mode)
if (available_credit > = required_credit)
        execute_virtual_I/O_request
else
        delay_I/O_request(CPUTime, SchedulingPolicy)
```

After delay:

```
increment := credit_by_control_mode(I/O device, current policy mode)
available_credit := available_credit + increment * (delayed_time / scheduling_period)
```

After processing I/O requests:

```
available_credit := available_credit - required_credit
```

Figure 6. Virtual I/O scheduling algorithm.

The available credit is reset at every scheduling period. If the current credit given to a guest domain is enough, the I/O request is executed immediately. If not, the amount of time to be delayed depends on the scheduling policy. After the delayed passes, the scheduler increases the credit score by the delayed time so as the delayed task to be executed again.

Rescheduling method should be also considered in the virtual I/O scheduler. In our para-virtualized Linux on Secure Xen on ARM, backend drivers handle requests in the bottom half context using tasklet, softirq, and workqueue, or in process context with kernel thread. For example, our network backend driver handles Tx requests from guest domains with a tasklet and the MTD backed driver handles I/O requests in a kernel thread per guest domain. Thus, to delay a request for a virtual device whose backend driver is implemented with tasklet, softirq, or workqueue, the virtual I/O scheduler cancels to preceed the current request and inserts the request again into a tasklet list, a softirq list, or workqueue after certain period of time using timer event, mod_timer(). When a kernel thread is used in a backend driver like MTD in our system, the kernel thread sleeps for a designated period of time and then wakes up to proceed the request.

3.4.2.4 ResourceAccount

The loaded regression formulas in the phase of virtual device initialization look like (1) and (2) in Section 3.2. ResourceAccount keeps track of each virtual resource usage of individual guest domain. The accumulated CPU time of Domain0 due to a guest domain's access to a virtual device is calculated using Equation3 in Section 3.3.

3.4.3 I/O Access Control Policy

As mentioned in 3.4.2, I/O scheduling is based on the access control policy, which states maximum capacity of a virtual device for each guest domain under several modes: normal, battery saving, and protection mode. A virtual I/O request is processed at normal rate at usual circumstances, but if the battery lifetime is under certain level, the virtual I/O scheduler starts to schedule requests with the battery saving mode policy to extend battery lifetime by reducing the unrestrained device use by a guest domain. When an important service in Domain0 needs to reserve a resource, virtual devices go into protection mode so that the important service could be continued without any interference or delay. If a denial of service attack is detected, yet isn't under battery limit, it is also possible to apply the protection mode to the virtual I/O devices. Note that an intelligent intrusion detection is beyond the scope of this paper.

```
<DomainPermission>
    <Subject>Domain1</Subject>
    <Permissions>
        <Type>vnetwork</Type>
        <Period>160000</Period>
        <NormalRate>100%</NormalRate>
        <BatterySaveModeRate>30%</BatterySaveModeRate>
        <-- BatteryLimit: the point to activate limited rate -->
        <BatteryLimit>40%</BatteryLimit>
        <ProtectionModeRate>10%</ProtectionModeRate>
    </Permissions>
    <Permissions>
        <Type>vmtd</Type>
        <Period>200000</Period>
        <NormalRate>100%</NormalRate>
        <BatterySaveModeRate>30%</BatterySaveModeRate>
        <BatteryLimit>40%</BatteryLimit>
        <ProtectionModeRate>20%</ProtectionModeRate>
    </Permissions>
</DomainPermission>
```

Figure 7. An example policy.

Scheduling period can be set differently among virtual devices in consideration of the fact that the average CPU time required to handle I/O requests are different among I/O devices.

XML format of I/O access control policy is provided for policy writers and is converted into binary for the virtual I/O scheduler. Our policy syntax in XML is shown in Figure 7. <*Period*> tag specifies the scheduling period of a virtual device in micro seconds. <*NormalRate*>, <*BatterySaveModeRate*> and <*ProtectionModeRate*> tags are the percentage representations from the value in <*Period*> tag. For example, 20% in <*ProtectionModeRate*> under the vmtd <*Type*> means that Domain1 is able to use 20% out of 200000 usecs of CPU time to access the virtual MTD device when it is under protection mode. <*BatteryLimit*> tag indicates battery lifetime to activate battery saving mode.

4. EVALUATION

In this section, we conducted experiments to evaluate the effectiveness of our fine-grained I/O access control against DoS attacks from a malicious guest domain. We measured the Domain0's CPU usage, the performance of I/O device access by Domain0, and the overall power consumption of an SGH-i780 in the following scenarios. First, we measure the above when there is no attack. Second, we execute a malicious program to simulate network attack in Domain1, which generates excessive outbound network packets, and then perform the same measurement. Third, we execute another malicious program in Domain1 which generates intensive NAND I/O requests, and perform the measurements for this case. Finally, we perform measurements when the fine-grained I/O access control is activated with a couple of I/O request control parameter values while the system is under DoS attacks.

4.1 Experiment Environment

This subsection provides a logical view of our experiment environment depicted in Figure 8 and Figure 9. Our testbed consists of three equipments: an SGH-i780 smart phone with our I/O access control modules, a Linux PC for remote controlling and measuring I/O device throughput of the smart phone, and a WT3000 Yokogawa precision power analyzer. An SGH-i780 is equipped with a 624MHz ARM9 processor, a 128MB DRAM, a 256MB NAND flash, and a 1480 mAh Li-ion battery. Two para-virtualized Linux 2.6.21 kernels are used for Domain0 as an IDD and for Domain1 on top of Secure Xen on ARM which uses 2MB of physical memory. 70MB and 56MB of the memory are allocated to Domain0 and Domain1 respectively. In addition, 50% of CPU share is assigned to each domain by the Xen Credit scheduler in work-conserving mode.

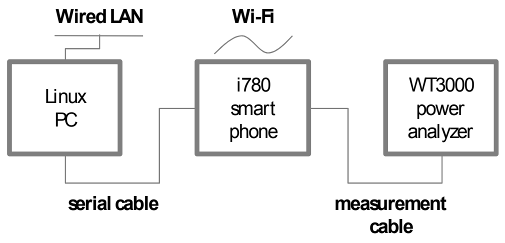

Figure 8. Construction of testbed

We placed voltage and current measurement probes between SGH-i780 body and its battery to obtain the power consumption of the SGH-i780. WT3000 was configured to probe power usages every 50msecs and to record the averages per 16 probing results in order to avoid unwanted effects from fast fluctuation of signal.

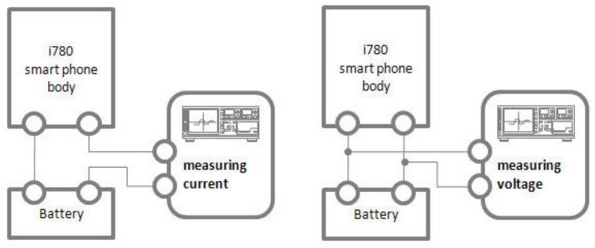

Figure 9. Wiring to measure current and voltage.

Figure 10 represents the software block diagram of the SGH-i780 smart phone and Linux PC. Programs used for our experiment are categorized into three groups.

The first group is comprised of attack programs simulating malware installed in Domain1: *net_atk* and *mtd_atk* belong to this group. The *net_atk* program executes UDP packet flooding attacks to impose heavy stress on the network device in the SGH-i780. The *mtd_atk* program produces overwhelming NAND READ operations to cause heavy load on NAND device in the smart phone.

The second group consists of measurement programs, *iperf* [22] and *Bonnie* [23]. *iperf* is a tool to measure maximum TCP bandwidth and UDP characteristics. *Bonnie* is a program to test the performance of file access with *read*, *write*, and *seek* operations.

The third group includes auxiliary programs, which comprise *minicom* and I/O ACM Policy Manager. *Minicom* is a communication program for console access of the smart phone through a serial port and I/O ACM Policy Manager is used to configure the usage limit of the I/O resources used by Domain1. In our experiment scenarios, we set I/O ACM control parameter value to 10% and 20% for each test case.

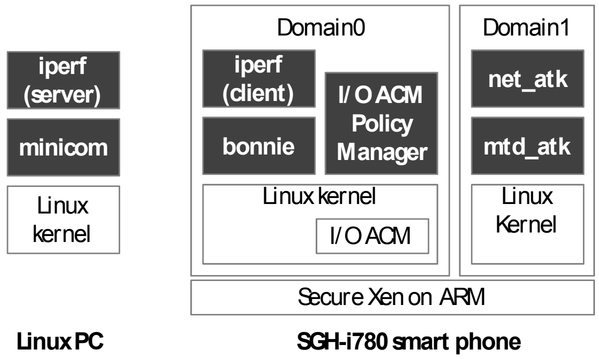

Figure 10. Software block diagram of i780 and Linux PC.

Packets generated by the *net_atk* program flow into a front-end network device driver in Domain1, then are transferred to a back-end network device driver in Domain0. Finally, these packets are handled by a native wireless network device driver in Domain0. This data flow is analogously applied when we run the *mtd_atk* program.

Table 2. Notations for test cases

	Network I/O Test Cases	Storage I/O Test Cases
No Attack	TcN0	TcS0
Under Attack (No I/O ACM)	TcN1	TcS1
Under Attack (20% I/O ACM)	TcN2	TcS2
Under Attack (10% I/O ACM)	TcN3	TcS3

The test cases are represented as 4-letter codes, “TcXY”. The prefix “Tc” means a test case. “X” indicates a device type, “N” for network and “S” for storage. “Y” represents a test condition 0-3. “0” means the 'normal' condition that there is no attack from Domain1. “1” indicates 'under attack' condition from malicious programs in Domain1 without fine-grained I/O ACM. “2” is the condition that the system is under attack where I/O ACM is enforced with the access control parameter 20%. “3” indicates the system is under attack where I/O ACM is enforced with the 10% usage limit.

We evaluate the CPU usage, performance overhead, and power consumption with these eight test conditions in the following sections.

4.2 Experiment Results: CPU Usage and I/O Throughput

For each test case, we measured Domain0’s CPU utilization with *xentop* [24], which is a utility program that displays information about each domain’s states. The *net_atk* program in Domain1 sends out UDP packets with the size of 44,160 bytes every 1000 usecs.

```
dom1.i780# net_atk –ip 10.0.2.1 –port 41002 &

dom0.i780# xentop
```

Similarly, we executed the *mtd_atk* program in Domain1, which scans every directory in the filesystem and reads file contents, with 3 different conditions: TcS1, TcS2, and TcS3.

```
dom1.i780# mtd_atk &

dom0.i780# xentop
```

Domain0’s CPU utilizations of the network and storage test cases are shown in Figure 11 and Figure 12 respectively. Basically, CPU usage in the normal condition (TcN0) is near zero. When the smart phone without enforcing I/O access control is under attack (TcN1), CPU usage peaks to 63%. Even though the maximum CPU share initially assigned to each domain is 50%, the TcN1 line shows that Domain0 takes over 60% of the CPU share to process the I/O requests from Domain1. However, CPU usage dropped to less than 30% by enforcing I/O ACM with the CPU usage limit of 20% (TcN2). I/O access control enforcement with the usage limit of 10% led to CPU usage drop as low as less than 10% (TcN3).

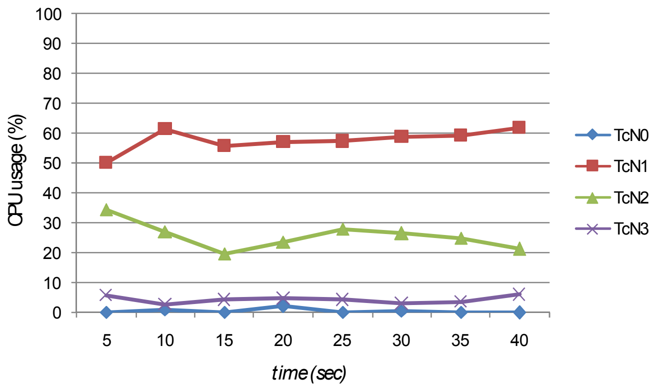

Figure 11. CPU usage in Domain0 (network I/O test cases).

Figure 12 depicts the effectiveness of our I/O ACM when it is applied to a storage device. TcS1 (Domain0’s CPU usage peaks to 90% due to the malware attack from Domain1) shows that it is possible for guest domains to attack IDD without any significant consumption of its own CPU time with a domain scheduler in work-conserving mode.

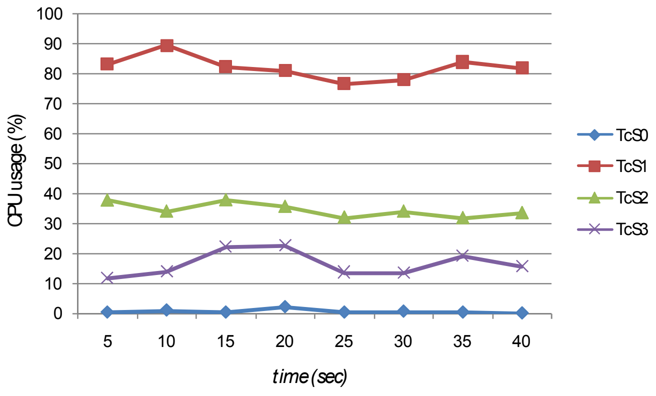

Figure 12. CPU usage in Domain0 (storage I/O test cases).

In addition to Domain0’s CPU usage pattern, we measured the throughput of network access and storage access with the 8 test cases. We launched *iperf* program with the following options to measure the UDP and TCP throughputs at the client side (SGH-i780). Each run lasts 10 seconds.

```
dom0.i780# iperf -c 10.0.2.1 -b 8M -f K -i 1 -l 1K -p 41001 -t 10 -u # client side

linuxPC# iperf -s -f K -i 1 -l 1M -m -p 41001 -u # server side
```

The commands to measure the TCP throughput at the client side (SGH-i780) are as follows:

```
dom0.i780# iperf -c 10.0.2.1 -f K -i 1 -l 1K -p 41001 -t 10 # client side

linuxPC# iperf -s -f K -i 1 -l 1M -m -p 41001 # server side
```

Table 3 and Table 4 show the results obtained from each experiment in terms of network and storage throughputs with the test cases.

Table 3. Network I/O throughput (on average)

Test Cases	UDP		TCP	
	Throughput [KB/Sec]	Ratio	Throughput [KB/Sec]	Ratio
TcN0	740.1	1	605.3	1
TcN1	209.2	0.2826	1.047	0.0017
TcN2	247.1	0.3338	193.5	0.3196
TcN3	496.6	0.6709	401.5	0.6633

Table 3 explains how much network throughput can be improved by enforcing our I/O ACM even under malware attack. First, UDP throughput of TcN1 (under attack and no I/O ACM) dropped to 28% compared to TcN0. When we enforced I/O ACM with the usage limit of 20% (TcN2), we could improve the UDP throughput to about 33.3% of TcN0. Furthermore, by setting the I/O access control parameter value to 10% (TcN3), we could obtain the improved UDP throughput up to 67% of TcN0's throughput. Second, the malware attack incurred a severe drop in TCP throughput due to the TCP retransmission mechanism. TCP throughput of TcN1 was 1.047 KBps, which is merely 0.17% of TcN0, where TCP packets are throttled by UDP packet flooding. However, after enforcing our I/O ACM, the TCP flow became stable and the throughput increased to 32% and 66% of TcN0's throughput by setting the usage limit to 20% (TcN2) and 10% (TcN3) respectively.

Table 4 shows the result collected through the four test cases from TcS0 to TcS3. In order to measure storage I/O throughput, we launched *Bonnie* with the following options.

```
dom0.i780# bonnie -s 4M # using a 4MB file for I/O performance measurement
```

The measured data consist of three items as follows: sequential output (KB/Sec), sequential input (KB/Sec), and random seek (number of seek operations). In case of TcS0, average throughput for sequential output was 1993.5KBps, but it dropped to 913.1KBps when there was an attack (TcS1's throughput: about 46% of TcS0's throughput). After enforcing the I/O ACM with the usage limit of 20% (TcS2), the sequential output throughput increased by 71% of TcS0's throughput. With the case of TcS3, the throughput increased by 86% of TcS0's throughput. Similarly, we could notice that the throughputs in both sequential input and random seek cases were improved by enforcing our I/O ACM as well.

Table 4. Storage I/O throughput (on average)

Test Cases	Seq.out		Seq.in		Rand.seek	
	Throu-ghput [KBps]	Ratio	Throu-ghput [KBps]	Ratio	Throu-ghput [KBps]	Ratio
TcS0	1993.5	1	4033.1	1	3931.9	1
TcS1	913.1	0.458	1649.6	0.409	1736.1	0.441
TcS2	1413.7	0.709	2902.7	0.719	3199.6	0.813
TcS3	1723.6	0.864	3597.4	0.891	3642.8	0.926

Through the experiments measuring both network and storage throughputs, we could observe that our fine-grained I/O ACM with a proper control parameter value effectively protected a smart phone against malware attacks from guest domains causing CPU overuse and performance degradation.

4.3 Experiment Results: Power Consumption

We finally measured the amount of overall power consumption in the SGH-i780 through the experiments with four network I/O test cases (from TcN0 to TcN3) and four storage I/O test cases (from TcS0 to TcS3).

Table 5 shows the result of power consumption in each I/O test case. In case of TcN0, the amount of power consumption was 1.9355W. When we launched the *net_atk* program in Domain1 without I/O ACM enforcement (TcN1), the amount of power consumption was 2.6005W (increased by 34%). On the other hand, if we enforced the I/O ACM with the access control parameter value of 20% under attack (TcN2), the amount of power consumption dropped to 2.3790W. With the access control parameter value of 10% (TcN3), power consumption decreased more.

In case of TcS0, the outcome of this experiment is almost the same as that of TcN0 because there was no actual difference in experiment environment except the time we performed each experiment. When we ran the *mtd_atk* program in Domain1 for a storage attack without I/O ACM enforcement (TcS1), the amount of power consumption measured was 2.3992W which is increased by 20% from the original reference value.

Table 5. Power consumption

Network			Storage		
Test Cases	Power (W)	Ratio	Test Cases	Power (W)	Ratio
TcN0	1.9355	1	TcS0	1.9354	1
TcN1	2.6005	1.3436	TcS1	2.3992	1.2396
TcN2	2.3790	1.2291	TcS2	2.1556	1.1137
TcN3	2.0916	1.0806	TcS3	2.0153	1.0412

When we enforced the I/O ACM with the access control parameter value of 20% (TcS2), the amount of power consumption decreased to 2.1556W. Furthermore, if we set the value to 10% (TcS3), we could get 2.0153W which was almost the same amount of power consumption measured in TcS0.

5. DISCUSSION

Although anti-malware solutions are widely used in PCs and mobile phones these days, they are not based on solid security mechanisms and are not a mission-critical system for the following reasons. First, they do not provide enough security against new malware propagation because they are insecure until the new malware vaccines are developed [25]. Second, they are inappropriate to achieve security guarantees since they may become compromised by the malware they are intended to detect [30]. As a result, they may not guarantee availability of the most important services of a smart phone (e.g., voice call, security service, etc) that should not cease regardless of circumstances. In order to solve these problems, we can run several domains for different purposes (e.g., Secure Domain for running important applications such as voice call and secure transaction, and Normal Domain for executing downloaded software [6]) on top of a VMM. By leveraging this architecture, our fine-grained I/O access control allows users to utilize the most important services (i.e., applications and services in the Secure Domain) in any circumstances even under new malware attacks generating excessive I/O requests from the Normal Domain.

Our approach effectively protects a mobile device against DoS attack from a malicious guest domain. From the experiments in Section 4, we could learn that there are strong correlation between the I/O device access, CPU usage, performance, and power consumption. Also, controlling precisely I/O requests from guest domains in terms of CPU usage prevents not only CPU overuse but also performance degradation and exhaustive power consumption. Although our access control is currently triggered by the remaining battery amount in our example policy shown in Figure 7, it could be more effectively used with an automatic reaction mechanism through the interaction of intrusion detection systems such as [26,27]. Despite many advantages, one drawback of our approach is that it is not scalable with the number of virtualized I/O devices in that every backend driver should be modified to support fine-grained control.

6. RELATED WORK

Deshane et al. [28] examine three virtualization environments—Xen, VMWare, and Solaris containers on PC/Server—to see how much overall performance degrades from simple stress tests on a virtual machine (VM). Their evaluation shows performance isolation cannot be guaranteed with existing virtualization technologies. Gupta et al. [20] propose a performance isolation mechanism across virtual machines in Xen. They noticed that the behavior of one VM can negatively impact resources available to other VMs even if appropriate per-VM resource limits are in place. By accumulating CPU usage per VM and changing the Xen scheduler, their approach provides performance isolation among VMs in a data center. However, their approach does not address security issues on a mobile device, but rather fair scheduling on a server system. Also, it limits the CPU usage per VM instead of controlling each device driver (i.e., suspicious device driver). As a result, it cannot provide fine-grained access control when a resource drain attack occurs through a certain I/O device.

Sailer et al. [29,30] present a secure virtualization architecture (sHype) by adding mandatory access control mechanism into Xen for the first time. Coker [31] provides a flexible security architecture (XSM) to add access control frameworks such as Flask [32] and it controls more system resources than sHype by putting various access control hooks in hypercalls [1]. Even though their approaches guarantee confidentiality and integrity of PC and server based on virtualization, they do not consider availability threat to mobile devices. Suh et al. [2,3,4,5,6] propose a secure virtualization technology (Secure Xen on ARM) for mobile devices by porting Xen to the ARM architecture and adding advanced security features fitted to mobile devices. Although it protects basic availability threat as well as confidentiality and integrity threats by supporting five appropriate access control models (i.e., Type Enforcement for controlling physical/virtual resources, Biba, Bell LaPadula for controlling virtual resources, Chinese Wall for controlling domain management, and Proprietary model for controlling CPU share of each VM), it does not cover I/O intensive attacks to an IDD by malware residing in a guest domain.

Garfinkel et al. [33] propose a virtual-machine-based platform for trusted computing, called Terra, that runs open-box VMs and closed-box VMs simultaneously. Although it provides confidentiality and integrity of the closed-box VMs which have critical applications (e.g., security apps.) by using attestation, it cannot guarantee reasonable performance and availability of the closed-box VMs when malware installed in an open-box VM tries to initiate a DoS attack to the closed-box VMs by sending extreme I/O requests.

Nash et al. [34,35] found out there is close correlation between processor usage and power consumption on an IBM Thinkpad T23 running Windows 2000 Professional. They also propose power estimation formula and malignant power attack detection method. However, they do not consider not only virtualization but also I/O excessive attacks.

7. CONCLUSION

This paper proposed a fine-grained I/O access control mechanism in an IDD to prevent DoS attacks from compromised guest domains. We find that due to the structural problem with IDD, excessive I/O requests from malicious guest domains could deteriorate performance of the entire mobile device and drain battery as well, despite Xen's fair allocation of system resources such as memory and CPU to each domain. The proposed fine-grained I/O access control mechanism in the IDD enables mobile devices to provide good performance of important services and applications, and to preserve durable battery life even under malware attacks from a compromised guest domain. By investigating the correlation between CPU usage and I/O requests from each guest domain, our mechanism provides accurate access control enforcement of I/O requests. We also perform experiments on CPU usage, performance, and power in three scenarios: normal, under attack, and access controlled. Our analysis shows our mechanism precisely controls CPU usage, and it provides good performance and power preservation under our access control enforcement, guaranteeing resources are available for critical services and applications.

Our future work is to incorporate an intrusion detection system with our fine-grained I/O access control mechanism in order to provide more effective protection mechanism to a smart phone.

8. ACKNOWLEDGMENTS

Our shepherd Brian Neil Levine gave us a great deal of valuable feedback that significantly improved the quality of this work. We thank the anonymous reviewers for suggestions that helped improve this paper.

9. REFERENCES

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," In Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP), 2003.

[2] S. Suh, "Secure Architecture and Implementation of Xen on ARM for Mobile Device," 4th Xen Summit, IBM T.J. Watson, April 2007.

[3] S. Suh, "Secure Xen on ARM: Status and Driver Domain Separation," 5th Xen Summit, Sun Microsystems, November 2007.

[4] Secure Xen on ARM project, http://wiki.xensource.com/xenwiki/XenARM

[5] J. Hwang, S. Suh, S. Heo, C. Park, J. Ryu, S. Park, C. Kim, "Xen on ARM: System Virtualization using Xen Hypervisor for ARM-based Secure Mobile Phones," In *Proceedings of the 5th Annual IEEE Consumer Communications & Networking Conference*, USA, January 2008.

[6] S. Lee, S. Suh, B. Jeong, S. Mo, "A Multi-Layer Mandatory Access Control Mechanism for Mobile Devices Based on Virtualization," In *Proceedings of the 5th Annual IEEE Consumer Communications & Networking Conference*, USA, January 2008.

[7] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson, "Safe Hardware Access with the Xen Virtual Machine Monitor," In *Proceedings of the Workshop on Operating System and Architectural Support for the on-demand IT Infrastructure (OASIS)*, 2004.

[8] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson, "Reconstructing I/O," Technical Report UCAM-CL-TR-596, University of Cambridge, 2005.

[9] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler, "An Empirical Study of Operating Systems Errors," In *Symposium on Operating Systems Principles*, pp. 73-88, 2001.

[10] P. Chubb, "Linux Kernel Infrastructure for User-Level Device Drivers," In *Proceedings of the Workshop on Object Systems and Software Architectures*, 2004.

[11] P. Chubb, "Get More Device Drivers out of the Kernel!," *Ottawa Linux Symposium*, 2004.

[12] F. Armand, M. Gien, G. Maigne, G. Mardinian, "Shared Device Driver Model for Virtualized Mobile Handsets," The Workshop on Virtualization in Mobile Computing, USA, June 2008.

[13] S. Sumpf, J. Brakensiek, "Device Driver Isolation within Virtualized Embedded Platforms," In *Proceedings of the 6th Annual IEEE Consumer Communications & Networking Conference*, USA, January 2009.

[14] Android Malware, http://www.wired.com/gadgetlab/2009/01 /rogue-googlepho/

[15] National Security Agency, "Security-Enhanced Linux (SELinux)," http://www.nsa.gov/selinux

[16] Trusted Computing Group, "TCG Mobile Trusted Module Specification," Specification version 0.9 Revision 1, 2006.

[17] W. A. Arbaugh, D. J. Farber, J. M. Smith, "A Secure and Reliable Bootstrap Architecture," In *Proceedings of IEEE Security and Privacy Conference*, pp. 65-71, 1997.

[18] Symantec Corporation, "SymbOS.Cabir," http://securityresponse.symantec.com/avcenter/venc/data/epoc.cabir.html

[19] A. Bose and K. G. Shin, "On Mobile Viruses Exploiting Messaging and Bluetooth Services," *Second International Conference on Security and Privacy in Communication Networks*, USA, 2006.

[20] D. Gupta, L. Cherkasova, R. Gardner, A. Vahdat, "Enforcing Performance Isolation Across Virtual Machines in Xen," In Proc. Of the ACM/IFIP/USENIX 7th International Middleware Conference (Middleware 2006), Australia, November 2006.

[21] Kprobes, http://www.ibm.com/developerworks/library/l-kprobes.html.

[22] *iperf*, http://sourceforge.net/projects/iperf/

[23] *Bonnie*, http://www.textuality.com/bonnie/

[24] *xentop*, http://www.xen.org/

[25] A. Joshi, S. T. King, G. W. Dunlap, P. M. Chen, "Detecting Past and Present Intrusions through Vulnerability-Specific Predicates," In Proceedings of the 20th Symposium on Operating Systems Principles (SOSP), October 2005.

[26] A. Bose, X. Hu, K. G. Shin, T. Park, "Behavioral Detection of Malware on Mobile Handsets," The Sixth International Conference on Mobile Systems, Applications, and Services, USA, June 2008.

[27] T. Garfinkel, M. Rosenblum, "A Virtual Machine Introspection Based on Architecture for Intrusion Detection," Internet Society's Symposium on Network and Distributed System Security (NDSS), 2003.

[28] T. Deshane, D. Dimatos, G. Hamilton, M. Hapuarachchi, W. Hu, M. McCabe, J. N. Mathews, "Performance Isolation of a Misbehaving Virtual Machine with Xen, VMware and Solaris Containers," White Paper, Clarkson University, February 2006.

[29] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. L. Griffin, and L. van Doorn, "Building a MAC-Based Security Architecture for the Xen Open-Source Hypervisor," *Annual Computer Security Application Conference*, 2005.

[30] R. Sailer, E. Valdez, T. Jaeger, R. Perez, L. van Doorn, J. L. Griffin, and S. Berger. "sHype:A secure hypervisor approach to trusted virtualized systems," IBM Research Report, 2005.

[31] G. Coker, "Xen Security Modules (XSM)," 3rd Xen Summit, , IBM T.J. Watson, 2006.

[32] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau, "The Flask Security Architecture: System Support for Diverse Security Policies," In *Proceedings of the 8th USENIX Security Symposium*, 1999.

[33] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, D. Boneh, "Terra: A Virtual Machine-Based Platform for Trusted Computing," 19th ACM Symposium on Operating Systems Principles (SOSP), 2003.

[34] D. C. Nash, T. L. Martin, D. S. Ha, and M. S. Hsiao, "Towards an Intrusion Detection System for Battery Exhaustion Attacks on Mobile Computing Devices," In *Proceedings of the Pervasive Computing and Communications Workshops*, pp. 141-145, 2005.

[35] D. C. Nash, "An Intrusion Detection System for Battery Exhaustion Attacks Mobile Computers," Master's thesis, Virginia Polytechnic Institute and State University, 2005.