Virtual Environments for Unreliable Extensions

A VEXE'DD Retrospective

Úlfar Erlingsson
Microsoft Research
Silicon Valley

Tom Roeder
Computer Science Department
Cornell University

Ted Wobber
Microsoft Research
Silicon Valley

June, 2005

MSR-TR-05-82

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Virtual Environments for Unreliable Extensions

A VEXE'DD Retrospective

Úlfar Erlingsson
Microsoft Research
Silicon Valley
Mountain View, CA
ulfar@microsoft.com

Tom Roeder*
Cornell University
Dept. of Computer Science
Ithaca, NY
tmroeder@cs.cornell.edu

Ted Wobber
Microsoft Research
Silicon Valley
Mountain View, CA
wobber@microsoft.com

ABSTRACT

We describe how virtualization techniques can be used to address the problems of reliability, security, and backward compatibility in extensible systems. We specify the conditions under which this approach can be applied and present an architecture for its implementation: VEXE, or *Virtual EXtension Environments*. Further, we detail our experience with implementing VEXE'DD, a system for increasing the reliability of Windows device drivers based on this architecture. This study extends, and puts into context, recent work on reliable extensibility mechanisms.

1. INTRODUCTION

This paper focuses on *extensible systems*, i.e., software applications or platforms whose functionality can be extended or modified with additional software. The software for these additions, or *extensions*, can take the form of scripts or machine-code, dynamically-loaded modules, libraries, standalone processes, or even compiled-in patches. In many extensible systems, and for many of these extension types, the goals of flexibility dictate that there are few restrictions on who can create or add extensions to the system.

Unfortunately, as well as enhancing the functionality of a system, extensions can greatly increase instability and the potential for flaws, and thereby reduce overall usefulness. This problem is exacerbated when the extensible system is a long-running software platform, such as a database, web server, or operating system, that serves multiple clients simultaneously. In these systems, a bug in a single extension (perhaps only used by one client) can force re-initialization of the entire system, along with loss of all client activity. This is a well-recognized practical problem for operating systems, where a device driver error can prevent the system from making progress or trigger fail-stop recovery requiring a system restart [2, 18].

The research described in this paper was done in the context of Microsoft Windows and motivated by the pervasive support for extensibility present in Microsoft products, and the wealth of existing third-party binary extensions that make use of that support. On a popular commodity system like Windows, reliability may hold few commercial rewards for third-party creators of extensions. As a result—and because of the sheer number of extensions, their varied quality, and the fact that they typically, for performance reasons, run without isolation in the address space of the extensible system—the instability caused by unreliable extensions is a particularly acute problem on Windows.

*Participated in this work while at MSR Silicon Valley.

Our approach makes use of virtualization in general, and virtual machines in particular, to allow increased reliability in the continued use of existing binary extensions. VEXE, or *Virtual EXtension Environments*, is an architecture that embodies our approach. In practice, backwards compatibility is necessary in any widely deployed system; our work on VEXE aims to satisfy that need and, thereby, provide a foundation for the future adoption of the new, reliable extensibility frameworks that are the subject of several current research efforts [12, 13, 15].

The design of VEXE builds on two fundamental observations about distinguished extensible systems.

- An extension cannot affect system reliability if it is executed, or hosted, by a separate, isolated instance of the extensible system, such as a virtual instance of the system.
- A reliable, loosely-coupled clerk component can safely mediate access to the services provided by extensions that are isolated in this manner.

Although these observations are simple, they form a foundation for the reliable use of existing extensions, even those of such disparate types as web browser plugins, image processing filters for graphics applications, and operating system device drivers.

VEXE'DD, or *VEXE for Device Drivers*, is an instance of our extension architecture that attempts to increase the reliability of unmodified x86 binary device drivers on Windows [7, 20]. VEXE'DD encapsulates unreliable device drivers in virtual instances of the Windows operating system that are created using a modified version of Microsoft's Virtual PC virtual machine software [16]. These modifications are necessary to allow drivers access to physical hardware and to related I/O functionality such as direct memory access. They also support the communication between a driver running in the virtual machine and another "master" operating system (and even the potential access to undefined structures in user-mode memory of that master). Also, they can allow multiple VEXE virtual machines to be established without consuming too many system resources.

Our implementation work on VEXE and VEXE'DD took place in 2003 at Microsoft Research, Silicon Valley. For device driver extensions, our implementation effort was stymied by the difficulty of the making the required modifications to our underlying virtual machine software. Because Virtual PC implements a hosted virtual machine monitor that is optimized to co-exist with an operating system (see Section 4.1)—and, therefore, doesn't have full, exclusive hardware control—it proved an inappropriate platform for our VEXE'DD implementation (no pun was originally intended). Since then, several publications have appeared that describe the isolation and recovery of device drivers, and the use of virtual machines [10, 14, 23, 24]. The retrospective study in this paper elucidates the principles behind the general approach, and these principles both clarify, and put into context, this recent work.

This paper is organized as follows. Section 2 gives more details on extensibility, defines some relevant terminology, and describes some of the problems raised by extensions in modern systems. Section 3 describes the design of our VEXE approach, its applicability and guarantees. Section 3 describes VEXE'DD: an implementation of the VEXE design in the context of Windows device drivers. Section 4 describes the Microsoft's Virtual PC software, the challenges we encountered implementing VEXE'DD, and some potential optimizations. To conclude, Section 5 discusses recent related work and Section 6 offers a retrospective on our work.

2. DEFINING EXTENSIONS

Extensions come in many shapes and sizes. Some examples include drivers for hardware, modules to extend operating systems with features such as encryption, content plugins to correctly render different media formats, Java web applets with stand-alone functionality, data-specific processing modules (such as shaders and filters in graphics applications), and server-side processing modules (such as cgi-bin code and DBMS stored procedures). In addition to the above, entire applications, such as word-processors or web-browsers, may be used as components or extensions by other applications. Furthermore, some systems employ pervasive loading of extensions to facilitate debugging or monitoring system-wide, or to offer a customized look-and-feel.

Due to the wide variety of contexts in which they are employed, it is hard to construct a single taxonomy for extensions. The Vino project made one of the few attempts at such a categorization by dividing operating system kernel extensions into three classes by their purpose: performance enhancement, policy modification, and functionality augmentation [19]. Instead of defining extensions in terms of such functional aspects, we focus on their structure and how they interact with the systems they extend.

We use the term *extension container* to describe the software into which extensions are added. Thus, an extension container could equally well be a stand-alone application or a general-purpose platform, such as an operating system. Extension containers can *encapsulate* extensions of multiple types (such as scripts and libraries) and in multiple ways (such as by dynamic linking or through IPC/RPC to another process [5]). We use the term *inproc extensions* to refer to those extensions implemented by binary machine code that is linked into, and given access to, the entire address space of the extension container.

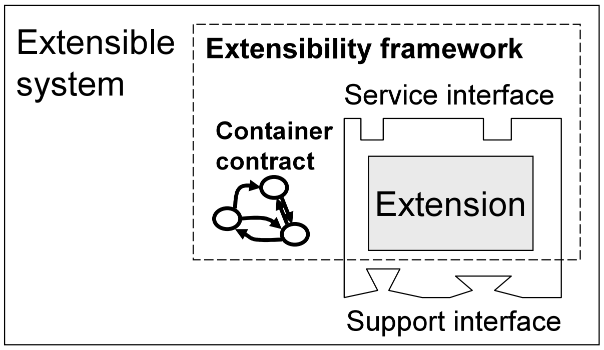

Figure 1: An overview of a typical extensibility framework; both control and payload data are passed accross the service interface.

An extension container defines an *extensibility framework* that aims to regulate, or impose structure on, the activity of extensions. (There are exceptions. Source code patches, for instance, are often outside of any framework.) In general, an extension interacts with its container through one of two interfaces: a *service interface* that characterizes the functionality the extension is adding, and a *support interface* that allows the extension access to resources and functionality of its container. Using support interfaces, for instance, extensions can allocate memory and access shared resources. Support interfaces can also provide access to abstractions present in the encapsulating container, such as web pages in the case of web-browser extensions.

An extensibility framework defines a model for how extensions use their interfaces. A container may define multiple such models, or *container contracts*, if that container supports multiple types of extensions. For instance, a web browser's contract with binary plugins may differ significantly from its contract with applets or embedded scripts. Container contracts typically impose requirements on how extensions provide service. Thus, a contract for extensions that decode media formats into bitmaps may specify who allocates the memory for that bitmap, as well as whether the encoded input data can be mutated and who must free that input memory.

For many common types of extensions, there is a large difference in how extensions and their containers interpret different types of data. Some values that pass across the service interfaces constitute *service control data*, which must conform to the container contract to ensure correct operation. Other values, however, are *service payload data*, which remains uninterpreted and has no bearing on an extension's compliance with its container contract. Thus—for the purposes of the reliable execution of its container—an extension for decoding image formats is not restricted in what bit patterns appear in its output bitmap.

This last observation is important: corrupt payload data cannot cause the incorrect operation of a container's extensibility framework—even if, eventually, some other activity depends on the correctness of that payload data. Therefore, despite certain types of extension flaws, a very real form of reliability is possible as long as the control data conforms to the container contract. Of course, the consequences of corrupt payload data may be more severe than an incorrect decoding of a bitmap image; for instance, applications may crash if the extension for an operating system's block or network devices returns corrupt payload data buffers. But, even in this case, the operating system should be able to operate correctly and provide service to other applications.

2.1 Ad hoc and Legacy Extensions

In practice, it can be difficult to describe an extensibility framework precisely. There may be only a modest distinction between support and service interfaces, and an even fuzzier boundary between service control data and service payload data. For inproc extensions, this uncertainty is compounded: these extensions can potentially affect their container in arbitrary ways and, therefore, their programmers must bear the burden of identifying and adhering to the proper container contract.

These practical issues create a slippery slope and, as a result, many extensions fall into the class of *ad hoc extensions*. Ad hoc extensions are characterized by reliance on implementation details of the encapsulating extensible system; thus, for ad hoc extensions, the implementation is the specification of the container contract.

Often, ad hoc extensions only passively rely on the implementation details of their container, e.g., by reading information from undocumented fields of data structures, or by making assumptions about the lifetime of data buffers they share with the container. For instance, assumptions about the ordering of Windows GUI events are common even when this order is not specified. Some ad hoc extensions go one step further, and actively make use of undocumented properties of their container, e.g., to directly modify otherwise inaccessible data structures. Even in this active form, however, ad hoc extensions typically aim to implement a recognized type of extension with an existing container contract—the passive and active assumptions are simply shortcuts to expedite the implementation, or overcome limitations of the contract. Only a few ad hoc extensions implement a truly new type of extension that is outside anything supported in the extensibility framework.[1]

New versions of extensible systems, in many cases, directly support functionality in the extensibility framework that has previously only been accessible to ad hoc extensions—and such direct support is especially likely when several popular extensions have made repeated use of the same ad hoc extensibility pattern. Thus, ad hoc extensibility can actually be advantageous for the vendor of a general-purpose commercial software platform: there is no need for any given version of the extensibility framework to completely support all possible uses, as long as there is sufficient possibility for extensions to achieve their goals through ad hoc means; the next version of the container contract can include whatever additional functionality turns out to be important in the marketplace. The opposite approach, of strictly enforcing isolation and a narrow container contract, runs the commercial risk of reducing the platform's applicability.

[1] Examples of extensions outside existing container contracts include some security software, such as anti-spyware, and extensions for on-the-fly compression of main memory. Amusingly, for our work, hosted virtual machines like Virtual PC are another example, as we describe in Section 4.1.

Of course, versioning of the extensibility framework creates problems of its own by defining a class of *legacy extensions*: those extensions that were created for previous generations of the container contract. Economic reality typically dictates backward compatibility with legacy extensions, and this can be complicated—especially since legacy extensions often make use of ad hoc extensibility. Because of this, new versions of extensibility frameworks are often required to preserve implementation details of the previous version. For instance, a large fraction of older Windows GUI software only executes correctly on today's versions of Windows because of a database of "application compatibility flags" that changes the delivery order of Windows GUI events to satisfy the ad hoc extensibility assumptions of those applications.

The existence of ad hoc and legacy extensions is not a historical accident, but a practical reality. Both classes have large consequences for the reliability of an extensible system by increasing its complexity and creating fragile dependencies on its implementation. This suggests that support for unreliable extensions, such as that of the VEXE approach, is not simply a stop-gap solution to address a temporary problem, but will be necessary in future reliable systems. In particular, this support can be critical for device drivers: a class of inproc extensions that may depend on several ad hoc aspects of their encapsulating operating system.

2.2 Device Driver Extensions

Device drivers in operating system are a particularly troublesome type of extension that has long been an obstacle to the reliability of computer systems. Not only can the instability of a device driver immediately corrupt any other system activity—device drivers are usually inproc extensions, often legacy, with an atypical and unforgiving container contract [2, 18]. This framework makes drivers particularly prone to errors and, as explained above, leads down the slippery slope of ad hoc extensibility. To compound the problem, operating systems often conservatively halt at the first sign of a device driver error.

Container contract violations in device drivers are one of the most common classes of operating system failure [**?**, 2, 20].Such failures can be triggered by the single use of incorrect service control data, even in drivers for devices, such as audio, where the payload may be of less consequence. In addition, because they are inproc extensions, drivers can also arbitrarily corrupt system memory (though this happens less often, at least in Windows). With current driver development techniques, such failures will remain frequent.

It has been noted that as computer peripherals and consumer electronics merge, drivers are increasing in numbers and decreasing in quality. The device driver problem is not improving, but getting worse [2]. In addition, the writers of device drivers must now also consider security, as well as reliability, since each driver exposes new possible venues for attack. Therefore, device drivers constitute an excellent testbed for the VEXE approach.

3. THE DESIGN OF VEXE AND VEXE'DD

The VEXE design tries to balance the goals of extensibility and reliability. These goals are always in conflict, since an extension may not respect the constraints and invariants of its container, and by violating them can cause crashes. To satisfy the demands of ad hoc extensibility, we allow extensions to run in their original, unmodified environment. Then, to further satisfy the demands of reliability, we isolate this extended version of the environment both by standard memory protection techniques, and also by of providing support for loose coupling of the service interface between the extension and its container.

We define a *master container* to be the original extensible system. In VEXE'DD, the master container is a Windows XP kernel [20], but in other instantiations of VEXE it may simply be an application process. We further define a *virtual container* to be an environment created by VEXE that encapsulates the unreliable extension in a copy of the extensible system. VEXE'DD uses a virtual machine environment to create a virtual container that encapsulates a separate copy of the Windows kernel that runs the isolated extension.

3.1 VEXE Containers

An implementation of the VEXE design serves a master container and instantiates virtual containers for unreliable extensions. The right choice for the type of virtual container will depend on the nature of the extension and the environment in which the master container is normally run. The key requirements, however, are that it be memory-isolated from the master container, and have some way to communicate with it.

For example, Adobe Photoshop [1] is an image editing application that allows inproc filter extensions to modify regions of pixels. These inproc extensions have full control over the application, and this is undesirable for extensions that are not known to be reliable. In the VEXE architecture, a master copy of Photoshop might start another copy of Photoshop and execute a filter extension in that environment. Given a means for the master to communicate with the now isolated filter extension, standard operating system process isolation might prevent this extension from crashing the master.

The isolated extension still needs access to the service interface for the master container, so the nature of the communication between the master and virtual containers is critical to the reliability of the master. In VEXE, a reliable mediating extension, called a *safe mediator*, takes the place of the unreliable extension in the master container. The safe mediator cooperates with the virtual container and its extension to process requests from the master container. These requests pass through a virtual container component called the *shim* that receives requests from the safe mediator and turns them into calls to the isolated extension. Figure 2 shows an overview of the VEXE design.

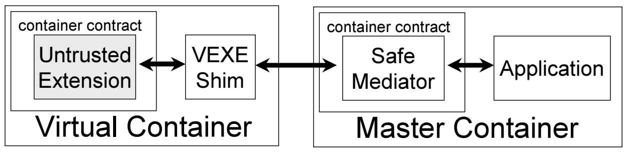

Figure 2: Overview of the VEXE architecture. Instead of directly using an unreliable extension, an extensible system, or master container, uses a safe mediator. Another copy of the extensible system, a virtual container, encapsulates the actual extension. The safe mediator uses the extension as an oracle, communicating with it using a VEXE shim.

VEXE'DD is an application of the VEXE design to device drivers. The master container in VEXE'DD is realized as the host operating system, and the virtual container is a virtual machine running a copy of that (or some other) operating system. The safe mediator is a kernel extension inserted into the host operating system that communicates with the extension in the virtual container.

The feasibility of isolated, remote device drivers, like those of VEXE'DD, is already demonstrated by several commercial products. Terminal Server [17] is an application for Microsoft Windows that allows a local user to start a remote desktop session on a networked server. Suppose that a Terminal Server user wants to play music using speakers connected to the local computer. In this case, it is pointless for the user's remote desktop session to play sounds on the server machine. Instead the Terminal Server implements a virtual sound card with a generic interface and forwards all requests to be played by the user's local client software. Failures of the user's local sound driver may inconvenience the user, but will, of course, have no effect on the stability of the server.

Similarly, in the implementation of USB devices in Virtual PC for Macintosh [16], the virtual USB device forwards all USB requests in the virtual machine out to its host's hardware USB device drivers. In this case, failure of the hardware will affect the virtual machine running on it. VEXE'DD reverses this design and runs the device driver in a virtual container where it cannot disrupt the host system.

The principal tenet of VEXE'DD is that drivers must be fully isolated and unmodified. Monitoring and associated recovery of the driver is insufficient as long as it can still access the system directly. Deciding for general device drivers which access patterns are acceptable and which are not is a complex task and is made much more difficult when the internal operations of the device driver can affect its host kernel. We require memory safety as well as control-flow safety; the device driver should not be able to touch any part of the kernel directly, and it should not be able to start executing any other unrelated part of the kernel. Because of this, and to support ad hoc extensibility, VEXE'DD uses virtual machines as virtual containers for device drivers.

Further, we need to isolate device drivers not only with respect to memory and control flow, but also in terms of calls to the service interface. To see why, consider the extreme case of isolation: we run the device driver on another machine (which happens to have the hardware that we need), and communicate with it across a network. In this case, the driver has no way of touching the memory of the kernel, yet it may still have unrestricted RPC access to the system. Using method calls, it can still destroy data structures, or otherwise cause crashes, by performing actions at the wrong time.

Such inappropriate calls, or corrupt control data, are as dangerous as arbitrary memory accesses, since they can violate the contract contract of the master container and its service interface. This problem cannot be solved by simple isolation. From the point of view of the operating system, the driver must conform to a state machine for the container contract. Such state machines allow the operating system to abstract away from the details of any particular driver, and treat large classes of devices equally; they also differ between operating systems (which explains why driver code is not easy to port).

Control data passed over a service interface directly affects this state machine. To prevent malformed control data from forcing the state machine to make an incorrect transition, a safe mediator must intercept, delay, and potentially modify any such control data.

3.1.1 *Assumptions about Failures*

VEXE'DD relies on several assumptions about the nature of device and driver failures. First, we assume that the hardware does not fail in a way that cannot be handled in software. If, for example, the display can be permanently damaged by the video driver, then software isolation cannot prevent such damage. Further, we do not interpose on the communication between the driver and and its device, so we cannot prevent malicious direct memory access.

On the software side, we either do not attempt recovery of failed drivers, or we assume that their failures are caused by transient states (perhaps due to timing problems between the driver and its device hardware). Since these transient errors are non-deterministic, they will not likely be present upon restart of the device driver, and so the system will ultimately be able to make progress. For some critical device drivers, such as the block device holding the system virtual memory pagefile, VEXE'DD may only be able to provide unclear guarantees.

3.2 Choosing an Abstraction Layer

In the VEXE design for unreliable extensions, the safe mediator can communicate with its isolated extension, through the shim, to any interface in the virtual container. This choice gives great flexibility to the VEXE implementor. On one hand, this communication may require only that the safe mediator pass calls across to the extension, validate and return its answers, and deliver callbacks from it. On the other hand, since the virtual container includes a full copy of the extensible system, the virtual container shim can be inserted at a higher layer of abstraction, if this is helpful. Thus, the safe mediator can be a driver extension in the master container, yet communicate with a shim that resides in a user-mode application in the virtual container. The use of such disparate interfaces, or abstraction layers, can simplify and increase the reliability of a VEXE implementation. In particular, the virtual container shim can be inserted at an interface that minimizes the need for a safe mediator to fully understand a complex container contract. In fact, by using interfaces at higher layer of abstraction, the safe mediator can avoid most of the details related to extension frameworks, and container contracts, even when those can still be causes of failures in the virtual container. Of course, the use of more abstract interfaces may come at a cost—either in terms of performance or functionality (e.g., user-mode interfaces may not support asynchronous operations). Thus, the implementor of a safe mediator and its shim can trade off reliability versus performance and functionality. In VEXE'DD, complex drivers whose contract is ill-understood are better handled within the virtual container, perhaps with a safe mediator that operates via user-mode interception in the master container.

The value of choosing the right abstraction layers can be clearly seen by returning to the previous Photoshop example. As before, a filter extension is isolated in its own virtual container, which consists of a separate Photoshop process. When the master Photoshop container, through its extensibility framework, attempts to use an unreliable filter extension to a given set of pixels, the safe mediator can be invoked. The safe mediator can take these given pixels, that were passed through the container contract, and write them to a file. Then, the safe mediator can ask its shim to open a new Photoshop virtual container with this new file, apply the filter extension, and then save a result file with the processed pixels. Finally, the safe mediator can read in this result file, and replace the original pixels with that file's contents—taking care not to overflow any buffers, or violate the Photoshop container contract in other ways. Thus, useful isolation has been achieved that leaves little possibility for the unreliable filter extension to affect the master Photoshop container.

Similarly, consider the implementation of virtual devices in Virtual PC 2004 [16]. Many of its virtual devices, even those of low-level hardware devices, are implemented as part of the Virtual PC application that runs in user mode on a host Windows operating systems. For example, when code in a guest virtual machine tries to write a block to disk, the virtual disk in this user-mode application receives the write requests, and makes use of user-mode Windows programming interfaces to write the block to a regular file that represents the virtual disk. Thus, user-mode Windows services are used to implement what is essentially the lowest-level of hardware operation. communicating directly with the hardware; a higher layer of abstraction has been used to great advantage.

3.3 The Oracle Model of Reliability

To protect the master container from crashing, the safe mediator must not allow calls to pass directly from the extension to the master container. We decouple the master container state machine from the unreliable extension by inserting the state machine of the mediator between them. The mediator knows the state machine of the master container, and will not allow calls to pass that might cause it to enter a state in which the only transition out is to crash. A couple of examples from Windows, described below, demonstrate these problems resulting from the unrestricted use of service interfaces.

The Windows kernel keeps track of the current Interrupt Request Level (IRQL) [18, 20]. Interrupt requests are mapped to IRQLs, as are deferred procedure calls, the clock, and other system activities. When an interrupt is received, the kernel raises the IRQL to the appropriate level and calls the Interrupt Service Routine (ISR) to invoke the correct driver. The ISR, however, is forbidden from making certain calls: for instance, it cannot touch paged memory, and it cannot wait on any object. If it does so, the kernel will choose to crash to avoid deadlock. A naively isolated driver in this environment would nonetheless be able to make any call to the kernel service interface, including touching paged memory or waiting on an object. Thus the safe mediator must intervene and not allow these calls to be made. If such calls are made, it must restart the virtual container and cause it to receive the interrupt again. This may cause some latency, but it is preferable to crashing the system.

Another example that occurs in Windows device drivers involves non-idempotent calls. A I/O Request Packet (IRP) is received by a driver when an application performs an IOCtl directed to that driver. The IRP contains both control and payload data, and is passed down the chain of drivers as the request is serviced. The Windows kernel provides a service interface, a “completion routine”, that must be called exactly once per IRP by device drivers. One common bug in Windows drivers is the failure to call, or the repeated invocation of, this completion routine. Either is fatal [2]. A safe mediator can maintain the system invariants pertaining to completion routines.

The essence of safe mediation is the use of a virtual container as an oracle to answer questions from the master container. Calls are forwarded from the master container to the shim, which then translates them to calls on the extension. We call this design the *oracle method*. The actual state machine of the extension need not be understood by the mediator; it only needs to understand the master container and forward its requests to the virtual container, checking any responses for safety. To be safe, the mediator does not need to understand how the extension wants to interact with its container. All the safe mediator requires is a model of the well-defined service interface of the master container. Thus VEXE eliminates the requirement to understand any ad hoc extensibility while still allowing ad hoc extensions to function.

For extensions that are device drivers, the state machine in question is that of the operating system kernel, and this can be very complicated. However, constructing this state machine is a job that can be performed once for a given OS version, rather than once per driver.

3.4 Recovering from Failures

Fortunately, critical devices that are fundamental to system operation are few and far between. Furthermore, their device drivers are normally written by operating system vendors and are more likely to be reliable. We do not currently consider isolating drivers for such devices.

A device driver can fail in a manner that causes it to incorrectly manipulate its hardware device. This may be fatal for devices like hard disks, on which critical data is stored. In other cases, such as for multimedia drivers, failure of the driver may mean that the screen is drawn incorrectly, but restarting or replacing the driver may, in this case, fix the problem. In fact, Windows already provides a simple demonstration of such recovery: in cases where the system can detect a video driver failure, it automatically switches to the VGA driver. This is a generic and well-tested driver, but offers only limited functionality. If the system had a safe way of restarting the failed driver, it might be able to retain full functionality (even without resetting its virtual container).

Reinitialization of a device driver to full functionality may in fact be possible. The operating system and the majority of drivers are written to correctly handle the removal and reset of their associated hardware device. For such devices, operating system container contracts allow for “unexpected device removal” to happen at any point [18]. So, a general solution in the case of device state corruption is to pretend such a removal and reset has occurred.

However, if a device driver, or extension in general, corrupts its virtual container (for instance by violating the container contract), there are few remedies. To recover from this case, it is necessary to restart the virtual container, after which the safe mediator can allow the device driver to reinitialize the hardware as if it were just starting up. This may cause momentary glitches, e.g., in network traffic or on the video display, but it is preferable to loss of the device. After this virtual container restart, the safe mediator can further initialize the device driver into a state that represents the master container’s viewpoint at the time of failure [23].

4. IMPLEMENTATION CHALLENGES

Our implementation of VEXE’DD was eventually brought to a halt by the amount of modification required to the underlying Virtual PC software. The most complex unreliable extension ever isolated by our implementation was a simplified version of the Windows floppy disk driver.

Even so, in our implementation effort, we designed techniques that address several interesting challenges to realizing the VEXE architecture for device drivers. Some of these techniques are specific to Virtual PC; others are generally applicable to any implementation for isolating drivers and safely mediating their interaction with a master operating system. This section details these challenges and techniques, but starts by describing Virtual PC itself, to set their context.

4.1 An Overview of Consumer Virtual PC

Our VEXE’DD implementation effort built upon a consumer version of Microsoft’s Virtual PC hosted x86 virtual machine monitor (henceforth VPC) [16] for Windows XP [20]. VPC is not a full hypervisor in the sense of [6, 11, 26]: it does not virtualize the hardware interface for all executing software, but rather shares control of the hardware with a master copy of Windows. This structure, a hosted virtual machine monitor (or VMM), is similar to that described by Sugerman et al. [22] and shown in Figure 3.

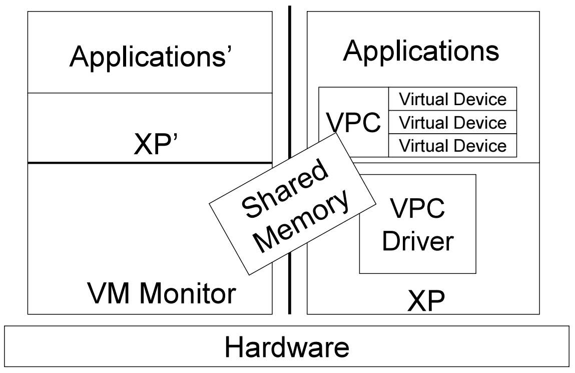

Figure 3: The three principal components of VPC: a user-level hardware emulator application, a kernel-mode extension, and a separate domain of hardware control, with a simple VMM, where the instructions of a guest virtual machine are executed.

VPC comprises three main components: (1) a user-mode application that handles interaction with the environment and the user, and emulates virtual hardware devices such as the disk to the Programmable Interrupt Controller (PIC); (2) a VMM [6, 11] that executes the instructions of a *guest virtual machine*, using ring compression and emulation for non-virtualizable x86 supervisor instructions; (3) and a kernel extension in the master operating system that transfers control to and from the VMM. Even though the user-mode application is responsible for updating most virtual machine state, and maintaining its invariants, this state (e.g., that of the virtual PIC), is visible to all three components through virtual memory mappings. Control is transferred to the VMM upon an IOCtl to the kernel extension from the user-mode application and, thus, the master operating system's scheduling of this application determines the schedule of the VMM's control of the hardware.

VPC uses a simple VMM that in most exceptional circumstances, such as traps or the receipt of an external interrupt, transfers control back to the master operating system. However, for performance reasons, the VMM includes a small emulator that sometimes effects atomic virtual hardware operations with a sequence of non-atomic instructions. These two aspects of the VMM can interact in complicated ways: for instance, if a hardware interrupt arrives during non-atomic VMM emulation, then eventually (perhaps before further VMM progress), the user-mode application may signal an interrupt on its virtual PIC—but, in this case, this virtual interrupt must not be signaled until the VMM completes the non-atomic emulation. In such scenarios, the components of VPC are required to collaboratively maintain the necessary invariants. As explained below, the complexity imposed by maintaining these invariants was a primary obstacle to our VEXE'DD modifications.

4.2 Implementing VEXE for Device Drivers

In VEXE'DD, we designed techniques for efficiently creating virtual containers that comprise a full operating system. We also developed mechanisms for dealing with the interactions between hardware and the VMM, operating system, and device driver software. These mechanisms address the identification of hardware resources, low-latency response to device interrupts, programmed and memory-mapped I/O, direct memory access by device hardware, as well as the unstructured sharing of memory between a kermel-mode driver and user-mode applications with special knowledge about that driver. These techniques and mechanisms are detailed below.

4.2.1 Identifying and Capturing Device Resources

To support driver extensions that control hardware devices, VEXE'DD must correctly identify the resources used by these, such as interrupt lines and I/O addresses. This identification is accomplished in the master container by intercepting key parts of the hardware discovery in Windows XP, e.g., PCI bus enumeration. In the master container, these resources are excluded from further use; in the virtual container, the VMM maps virtual resources to the physical resources identified. Then, when the master container uses those resources for driver instantiation, a safe mediator is created instead, and the unreliable driver extension is instantiated in the virtual container.

4.2.2 Interrupts from Devices

When a hardware device signals a physical interrupt, there are two possible cases: either the VMM or Windows XP may be in control of the hardware. If the VMM is in control, a flag can be set to indicate that the VMM should, as soon as possible, return control back to the VPC user-mode application so that it can signal a corresponding interrupt in the virtual PIC. This flag allows any non-atomic emulation to be completed, and the VPC application will maintain all necessary shared state invariants. At the hardware level, the interrupt may be masked, or disabled, as soon as the flag has been set; only when the virtual interrupt line is enabled does the physical interrupt line need to be unmasked.[2]

If, however, the interrupt happens while Windows is running, VPC may not be the currently scheduled application. Therefore, when it is notified of the interrupt, our modified VPC kernel extension faces a dilemma. It must either wait for or force the VPC application to be scheduled—and things can progress much as above—or do something more clever. While waiting for scheduling quanta measured in milliseconds may be acceptable for latency-tolerant devices like the floppy controller, it is not an acceptable general solution. Forcing an application context switch on each interrupt, with the associated latency and scheduling artifacts, is similarly unappealing.

Hence, in our VEXE'DD design, we use an approach motivated by Software Fault Isolation (SFI) [25]. We extract all the machine code that may possibly run in the interrupt handler of the unreliable driver extension, including the code for all support and service interfaces that are called. In this code, we use instrumentation, or machine-code rewriting [21], to change the semantics of all instructions that access memory. The new semantics redirect all memory accesses to the correct physical memory pages (i.e., the right

[2]Here, we do not detail the differences needed for shared and edge-triggered interrupts, which are both special, e.g., in the delivery of the EOI command to the physical PIC. Nor do we describe affinity or other symmetric multi-processor systems issues.

virtual container memory) even when the code is executed in the master container. Thus, the virtual container's physical memory must be mapped by the kernel-mode page tables of the master Windows XP container; this mapping is easily done by our modified VPC kernel extension.

This technique can allow rewritten unreliable interrupt handlers to be executed within the master container without being able to affect any state in that container. The code to be instrumented can be identified either statically, from driver binaries, or dynamically from the pointers assigned to the virtual interrupt descriptor table. For Windows device drivers, the instrumented code will not comprise very many x86 machine-code instructions: WDM drivers must return quickly from interrupts, and schedule any non-trivial processing to a later "deferred procedure call". Of course, any privileged x86 instructions, such as those that touch machine-specific registers, must be modified to only affect the state of the virtual container or throw an exception. On such exceptions—as well as on a computed jump to code that was not instrumented, the access of a virtual container address not pinned to physical memory, or the expiration of a timeout limiting the execution duration of instrumented code—the processing can be cut short, with the interrupt masked, and normal VPC operation will eventually resume from this point. By "giving up", correctness is maintained, at the cost of reduced performance.

When processed in this manner, the interrupt-handler code of unreliable drivers can also be executed within the context of the VMM, as long as care is taken to maintain the invariants the VMM shares with other components of VPC. Furthermore, we believe that this SFI-inspired technique is generally applicable not just to hosted VMMs, but to all hypervisor systems that require low-latency interrupt response.

4.2.3 Driver Programmed I/O

Once a driver receives an interrupt, it normally must get data from the device. In many drivers, some communication with the hardware device is done by programmed I/O—i.e., through executing x86 `in` and `out` instructions in the driver code that, respectively, read a data byte from and write a data byte to the device. The destination device for these instructions is specified using an I/O port number operand.

In VPC, normally the destination specified by port numbers will either be invalid or a virtual device in the VPC application; this is determined in VMM emulation of programmed I/O instructions using a sequence of non-atomic instructions. In VEXE'DD, we modified this sequence to directly access certain I/O ports: those of the unreliable driver's hardware device. (Another alternative would have been to change the x86 "I/O permission bitmap" [7].)

4.2.4 Memory-Mapped I/O and DMA

VEXE'DD must, of course, handle the other methods of communication between the device and the driver—memory-mapped I/O and DMA direct memory access from devices. Our modified VMM can address both because of its control over the virtual container's address space mappings. On one hand, the virtual resources used for driver instantiation are made to refer to device I/O register memory.

On the other hand, the modified VMM uses ad hoc techniques similar to "paravirtualization" [3] to change the addresses returned to driver DMA-buffer requests to be those of reserved physical memory allocated to the virtual container. The driver sends these physical addresses to the device—using commands opaque to VEXE'DD—making the hardware directly access that memory. (Note that, because of these opaque commands, we cannot protect against failures where the driver sends the incorrect physical address to its device. Such protection is possible only with additional hardware, although this is apparently forthcoming [10].)

4.2.5 Handling METHOD_NEITHER IOCtls

One mode of driver communication poses a substantial challenge to both VEXE'DD and other driver isolation approaches. In some situations, code with intimate knowledge of a driver will pass it pointers to data structures with no defined or documented structure. For Windows drivers, this is known as METHOD_NEITHER, and is often used in IOCtls from user-level configuration programs that are included in driver software distributions [18]. Communication via pointers with unknown semantics is also common in drivers for advanced graphics devices, and other code where with applications utilize devices through libraries supplied by the driver vendor.[3] (This is particularly unfortunate, since drivers for such cutting-edge devices are a frequent source of unreliability.)

There is a general technique for addressing this problem of opaque communication: changing where VEXE is implemented to an abstraction layer with well-defined semantics. For instance, the libraries of even the most state-of-the-art graphics card will expose standard interfaces (e.g., Microsoft's DirectX) to games and applications, and can be replaced with a safe mediator that exposes the same interfaces. Thus, the best implementation for VEXE'DD may not always be inside the Windows kernel, but in user mode applications. (For user-mode applications, another alternative would be to map the page tables for user-mode virtual addresses of the master container as read-only into the virtual container. This mapping might even allow a driver to write in certain ways to user-mode applications—although this would most likely conflate the failure of the driver and these applications.)

4.3 Optimization of Container Creation

It is important, in any implementation of the VEXE approach, that resource consumption be kept to a minimum. This is especially true in VEXE'DD: without any optimizations, the resource use of multiple concurrent operating system containers can easily overwhelm the system capacity, e.g., in terms of physical memory.

Fortunately, memory consumption can be reduced significantly in those cases where virtual containers are mostly the same. In VEXE'DD, if the same version of Windows is used in many containers, then most code pages will be read-only and contain the same bits; other pages may also hold the

[3] This type of interdependence between user-mode code and 3^{rd}-party device drivers can also be a source of security problems: in effect a device driver IOCtl is a new system call over which the operating system has no control, and whose semantics may introduce arbitrary vulnerabilities.

same data, and be never or rarely written to. This allows resource optimization using the techniques of content-based sharing [26], where only one copy of a given memory page can serve multiple containers.

Even with content-based sharing, VEXE'DD virtual containers may waste resources on tasks, such as general housekeeping, that are irrelevant to driver encapsulation. By relying on symmetric multi-processor support this wasted work can be reduced, and more state can be shared. As long as the master container can be started believing it has extra CPUs (that need not really exist) and a snapshot of its state can be taken just before that processor is activated, a template for lightweight virtual containers can be created. Content-based sharing will work well if all containers are derived from this template. More importantly, in the virtual containers, all non-essential interrupts and processes can be set to have affinity with the non-existent CPUs—and, thus, they will never incur any overhead.

5. RELATED WORK

There has been considerable prior work on extensible systems. Most notably, the SPIN [4], Vino [19], and SFI [25] systems have all made important contributions towards making extensions reliable. These systems do not, however, attempt to deal with ad hoc, legacy extensions which are the motivation and focus of this work.

From early on in operating system research, it was recognized that device drivers did not truly belong in operating system kernels—e.g., some proposals suggested that drivers should be run in intermediate protection rings [8]. User-mode Linux [9] takes this approach one step further, and runs kernel extensions (if not device drivers) in a Linux virtualized at the level of the system-call interface. Thus, extensions run as applications and process isolation separates these extensions from the kernel and from other applications. This technique usually requires recompilation, since kernel extensions do not expect to run in user mode. In addition, access to physical hardware is problematic.

Rather than seek a different abstraction for device drivers, Nooks [23, 24] leaves them as kernel extensions nearly completely in the protection domain of the kernel, but provides some narrowing of that domain by placing the driver in a memory isolation container known as a *nook*, and monitoring its service interface activity. By detecting failures, e.g., via timeout or memory protection faults, Nooks may catch many types of driver bugs—upon which, Nooks attempts to recover by re-initializing the driver with shadow state tracked during monitoring. Drivers must be ported to Nooks, and the nook must include all necessary support interfaces. This porting may prove difficult for complex drivers, and for many ad hoc, legacy drivers, it may be impossible.

The Xen [3, 10] virtual machine framework presents a different model for device driver isolation. In [10], the focus is on safely sharing access to device drivers, even across systems. This work clearly addresses some of the problems that motivated our work—in particular, the problem of ad hoc device drivers and their unfettered use of support interfaces. In Xen, a driver is typically isolated in its own I/O virtual machine, and connected to other kernels via a unified driver interface. Because Xen uses paravirtualization, use of this interface generally requires recompilation of the driver.

The system that most closely parallels our own is due to LeVasseur et al. [14]. In this work, device drivers run in containers: full copies of an operating system. These isolation containers are treated as device drivers and controlled through a translation module; each isolated driver may in turn call other such isolated drivers. Like Xen, this work makes use of paravirtualization. The main difference between this work and ours is how service interfaces are redirected through a simple wrapper—and not a safe mediator.

While the above recent work on driver isolation can be seen as validating the possibility of an VEXE'DD implementation, this work has primarily aimed to reduce the damage of unreliable extensions by changing their protection domain. Our work on VEXE suggests that a larger problem must be addressed—otherwise, the possibility remains for contracts to be violated and the operating system's critical state machines to be corrupted [2].

6. VEXE'DD IN RETROSPECT

In retrospect, our work on VEXE, and our thwarted implementation effort has proved enlightening. In particular, VEXE and VEXE'DD complements recent related work on reliable extension mechanisms by clarifying the following important points:

- The existence of an important class of ad hoc, legacy extensions that closely depend on implementation details is a strong argument for isolating extensions using virtual machines, or other virtual containers.

- For an extension to be reliable, it must necessarily always conform with the container contract; an unreliable extension can be made to satisfy this requirement through the use of a safe mediator based on the oracle model.

- Although some container contracts are not well suited for safe mediation, e.g., because of their complexity, there is often room for reliably supporting extensions at a different abstraction layer.

- The VEXE'DD effort demonstrates that the details of virtual machine techniques, such as paravirtualization and hosted VMMs, have important practical implications on the applicability of virtual containers to low-level extensions like device drivers

- Finally, as our work shows, many novel, efficient implementation techniques are possible for optimizing performance, e.g., for low-latency response to device interrupts and the lightweight creation of multiple related virtual containers.

Hopefully, VEXE and similar approaches will allow future extensible systems to be simplified by removing support for backward compatibility and ad hoc, legacy extensions. If this happens, the door may be open for future, truly reliable extensibility frameworks.

Acknowledgments
This work benefited greatly from repeated interaction with Paul Barham. Chats with Dave Conroy, Mike Burrows and Emin Gün Sirer were inspiring. Thanks to Eric Traut, Jake Hoelter, and others from the VPC group, as well as to Byron Cook, John Richardson and Peter Wieland from the Windows base team. Kevin Walsh commented on a previous version of this paper.

7. REFERENCES

[1] Adobe Systems Inc. Adobe Photoshop, 2005. `http://www.adobe.com/products/photoshop/main.html`.

[2] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and Static Driver Verifier. Technical Report MSR-TR-2004-8, Microsoft Research, Jan. 2004.

[3] 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 *Proc. SOSP'03*, pages 164–177, 2003.

[4] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. J. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proc. SOSP'95*, pages 267–284, 1995.

[5] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.

[6] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *Proc. SOSP'95*, pages 1–11, 1995.

[7] B. B. Brey. *The Intel microprocessors: architecture, programming, and interfacing.* Prentice Hall, 1997.

[8] R. C. Daley and J. B. Dennis. Virtual memory, processes, and sharing in MULTICS. *Commun. ACM*, 11(5):306–312, 1968.

[9] D. Frascone. Debugging kernel modules with user-mode Linux. *Linux J.*, 2002(97):5, 2002.

[10] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *OASIS Workshop at ASPLOS'04*, Boston, MA, October 2004. `http://www.cl.cam.ac.uk/Research/SRG/netos/xen/architecture.html`.

[11] R. P. Goldberg. Architecture of virtual machines. In *Proc. of the workshop on virtual computer systems*, pages 74–112, 1973.

[12] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proc. OOPSLA'03*, pages 388–402, 2003.

[13] G. C. Hunt and J. R. Larus. Singularity design motivation. Technical Report MSR-TR-2004-105, Microsoft Research, Dec. 2004.

[14] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proc. OSDI'04*, pages 17–24, San Francisco, CA, December 2004.

[15] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. In *Proc. ACM conf. on Language Design for Reliable Software*, pages 128–137, 1977.

[16] Microsoft Corporation. Microsoft Virtual PC 2004 technical overview, 2004. `http://www.microsoft.com/windows/virtualpc/evaluation/techoverview.mspx`.

[17] Microsoft Corporation. Windows 2000 Terminal Services, 2005. `http://www.microsoft.com/windows2000/technologies/terminal/default.asp`.

[18] W. Oney. *Programming the Microsoft Windows Driver Model.* Microsoft Press, 2 edition, 2002.

[19] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. OSDI'96*, pages 213–227, Oct. 1996.

[20] D. Solomon and M. Russinovich. *Inside Microsoft Windows 2000.* Microsoft Press, 3 edition, 1999.

[21] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, 2001.

[22] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *Proc. USENIX'02 Technical Conf.*, pages 1–14, June 2001.

[23] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *Proc. OSDI'04*, pages 1–16, San Francisco, CA, December 2004.

[24] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *Proc. SOSP'03*, pages 207–222, 2003.

[25] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. *SIGOPS Oper. Syst. Rev.*, 27(5), 1993.

[26] C. A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002.