Rethinking the Library OS from the Top Down

Donald E. Porter[†], Silas Boyd-Wickizer[‡], Jon Howell, Reuben Olinsky, Galen C. Hunt

[†] Department of Computer Science
Stony Brook University
Stony Brook, NY 11794

[‡] MIT CSAIL
32 Vassar Street
Cambridge, MA 02139

Microsoft Research
One Microsoft Way
Redmond, WA 98052

"There is nothing new under the sun, but there are a lot of old things we don't know."

– Ambrose Bierce, The Devil's Dictionary

Abstract

This paper revisits an old approach to operating system construction, the *library OS*, in a new context. The idea of the library OS is that the personality of the OS on which an application depends runs in the address space of the application. A small, fixed set of abstractions connects the library OS to the host OS kernel, offering the promise of better system security and more rapid independent evolution of OS components.

We describe a working prototype of a Windows 7 library OS that runs the latest releases of major applications such as Microsoft Excel, PowerPoint, and Internet Explorer. We demonstrate that desktop sharing across independent, securely isolated, library OS instances can be achieved through the pragmatic reuse of networking protocols. Each instance has significantly lower overhead than a full VM bundled with an application: a typical application adds just 16MB of working set and 64MB of disk footprint. We contribute a new ABI below the library OS that enables application mobility. We also show that our library OS can address many of the current uses of hardware virtual machines at a fraction of the overheads. This paper describes the first working prototype of a full commercial OS redesigned as a library OS capable of running significant applications. Our experience shows that the long-promised benefits of the library OS approach—better protection of system integrity and rapid system evolution—are readily obtainable.

Categories and Subject Descriptors D.4 [**Operating Systems**]: Organization and Design.

General Terms Experimentation, Performance.

1. Introduction

The *library OS* approach to OS construction was championed by several operating system designs in the 1990s [3, 10, 13, 21]. The idea of the library OS is that the entire *personality* of the OS on which an application depends runs in its address space as a library. An OS personality is the implementation of the OS's application programming interfaces (APIs) and application visible semantics; the OS services upon which applications are built. Early proponents of the library OS approach argued primarily that it could enable better performance through per-application customization. For example, a disk-I/O bound application with idiosyncratic file access patterns can realize better performance by using a custom file system storage stack rather than using the default sequential prefetching heuristics.

Like many of its contemporaries, the library OS approach is largely forgotten, a casualty of the rise of the modern virtual machine monitor (VMM) [8]. While most new OS designs of the time—including library OS designs—ran only a handful of custom applications on small research prototypes, VMM systems proliferated because they could run major applications by reusing existing feature-rich operating systems. The performance benefits offered by library OS designs did not overcome the need for legacy compatibility. On the other hand, the need for security and independent system isolation has increased since the 1990s due to the rise of the Internet.

We revisit the library OS approach to OS construction, prioritizing application compatibility, security isolation, and independent system evolution benefits. Our goal is to realize the benefits of a VMM, but with order-of-magnitude lower overheads. This paper shows for the first time that it is possible to build a library OS running major applications from a feature-rich, commercial OS. We describe a Windows 7 library OS, called *Drawbridge*, running the latest commercial releases of a large set of applications, including Microsoft Excel, PowerPoint, Internet Explorer, and IIS. Windows applications running on Drawbridge have access to core Windows features and enhanced APIs including the .NET common language runtime (CLR) and DirectX.

This paper describes a new top-down approach to building library OSes—an approach that prioritizes application compatibility and high-level OS code reuse and avoids low-level management of the underlying hardware by the library OS. Drawbridge demonstrates that a small set of OS abstractions—threads, virtual memory, and I/O streams—are sufficient to host a Windows 7 library OS and a rich set of applications. This small set of abstractions helps simplify protection of system integrity, mobility of applications, and independent evolution of the library OS and the underlying kernel components. Despite being strongly isolated, Drawbridge applications can still share resources, including the screen, keyboard, mouse, and user clipboard, across independent library OS instances through the pragmatic reuse of networking protocols.

As a structuring principle, we identify three categories of services in OS implementations: *hardware services*, *user services*, and *application services*. Hardware services include the OS kernel and device drivers, which abstract and multiplex hardware, along with file systems and TCP/IP networking stack (see Figure 1). User services in the OS include the graphical user interface (GUI) shell and desktop, clipboard, search indexers, etc. Application services in the OS include the API implementation; to an application, these comprise the OS personality. Applications services include frameworks, rendering engines, common UI controls, language runtimes, etc. The application communicates with application services, which in turn communicate with hardware and user services.

ASPLOS'11 March 5–11, 2011, Newport Beach, California, USA.

We use these service categories to drive the refactoring of Windows into the Drawbridge library OS. Drawbridge packages application services into the *library OS* and leaves user and hardware services in the *host OS* (see Figure 2). The library OS communicates with hardware services in the host OS through a narrow *application binary interface* (ABI), which is implemented by a *platform adaptation layer* and a *security monitor*. The library OS communicates with user services in the host OS using the *remote desktop protocol* (RDP) [28] tunneled through the ABI. Each application runs in its own address space with its own copy of the library OS. The security monitor virtualizes host OS resources through its ABI with the library OS and maintains a consistent set of abstractions across varying host OS implementations. For example, the file system seen by an application is virtualized by the security monitor from file systems in the host OS.

Previous library OS designs aimed to provide application-customized performance enhancement, and thus exposed low-level hardware abstractions to applications. Cache Kernel, Exokernel, and Nemesis innovated by providing applications with fine-grained, customized control of hardware resources, such as page tables, network packets, and disk blocks [10, 13, 21]. In contrast, Drawbridge's differing goals (security, host independence, and migration) free it to offer higher-level abstractions. These higher-level abstractions make it easier to share underlying host OS resources such as buffer caches, file systems, and networking stacks with the library OS. By making low-level resource management an independent concern from OS personality, each can evolve more aggressively.

One can view a modern VMM as a mechanism for automatically treating a conventional OS as a library OS, but the facility incurs significant overheads. Each isolated application runs in a different dedicated VM, each of which is managed by a separate OS instance. The OS state in each VM leads to significant storage overheads. For example, a Windows 7 guest OS in a Hyper-V VM consumes 512MB of RAM and 4.8GB of disk. In contrast, Drawbridge refactors the guest OS to extract just those APIs needed by the application; it adds less than 16MB of working set and 64MB of disk.

The Drawbridge approach could significantly impact desktop computing by enabling fine-grain packaging of self-contained application. The VMM approach allowed the construction of self-contained application packages comprised of an application and OS, with minimal dependencies on the underlying VMM or hardware. VMM-based application packaging enabled a huge shift in server computing through server consolidation and cloud computing [1, 25], all despite huge overheads. We believe that the finer-grained, higher-performance application and OS packages that are now possible with library OSes could precipitate similar shifts in desktop and mobile computing; for example, snapshots of running Drawbridge applications could easily move from device to device and to the cloud because they are so small—a compressed process snapshot of Excel is under 4MB for Drawbridge, versus nearly 150MB for a similar VM snapshot.

In summary, this paper describes the first refactoring of a widely used, monolithic OS into a functionally-rich library OS. It contributes a set of heuristics for refactoring a monolithic kernel into a library as well as a new ABI for separating a library OS from host OS. The benefits of this design include: 1) strong encapsulation of the host OS from the library OS, enabling rapid and independent evolution of each; 2) migration of running state of individual applications across computers; and 3) better protection of system and application integrity (i.e., strongly isolated processes).

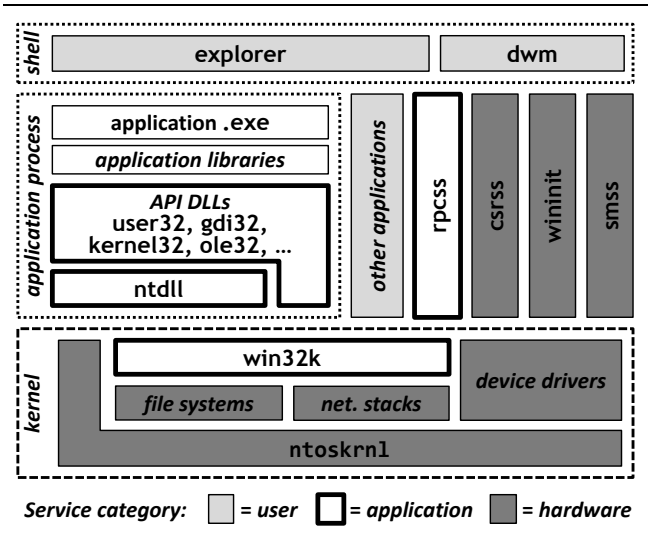

Figure 1. Windows 7 OS Architecture.

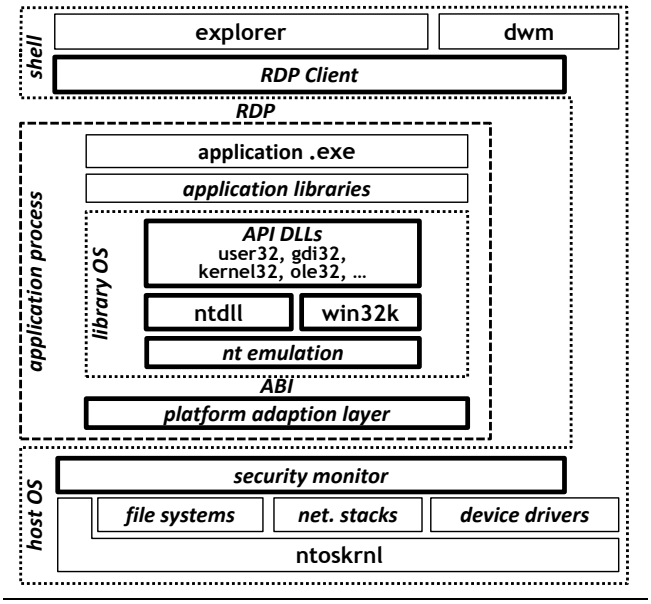

Figure 2. Drawbridge Architecture.

The remainder of the paper is structured as follows. Section 2 provides helpful background on the Windows 7 OS architecture as it exists prior to Drawbridge. Section 3 describes our approach for separating the Win32 personality from the rest of Windows to create a library OS. Section 4 describes the security monitor and the ABI it presents to the library OS. Section 5 describes our implementation of the Windows 7 library OS. Section 6 evaluates the scalability, complexity, update surface, and overheads of our approach through examination of the current implementation. Section 7 places Drawbridge into context with related work. In Section 8 we discuss possible impacts of the library OS approach on end-user computing. Finally, we summarize our contributions in Section 9.

2. Windows 7 Architecture

Refactoring a general-purpose, commercial operating system away from its long-evolved roots into a library OS is a significant challenge. This section outlines the key features of the Windows 7 architecture, drawing analogies between elements in Windows and corresponding elements common to such Unix systems as Linux. We draw these analogies to help the reader appreciate that similar challenges exist in other large-scale OSes.

Windows is architecturally divided into *dynamic link libraries* (DLLs) that load into an application's address space, *services* that run as daemon processes, the *NT kernel*, and *drivers*, which are dynamically loaded kernel-mode components (see Figure 1). DLLs are similar to shared object (.so) files in Unix systems. The NT kernel implements the usual core of a monolithic operating system: resource management, scheduling, I/O services, a hierarchical object namespace, and the *registry*, a key-value store for configuration data. Replaceable kernel-mode components, such as the networking stack and file systems, are loaded as drivers.

In practice, all Windows applications are programmed to the Win32 API. While there is no official count, Win32 is known to include over 100,000 API functions. The API is implemented as a large collection of in-process DLLs. These DLLs access the NT kernel indirectly through a runtime DLL, ntdll, which the kernel inserts into every process during creation. ntdll implements the dynamic loader for DLLs along with functionality roughly equivalent to the Unix libc, including stubs for the 401 functions in the NT system-call table. Commonly used Win32 API DLLs include: kernel32, which provides access to kernel operations for process control, threading, virtual memory, registry, and block and character device I/O functionality; user32 and gdi32, which provide the basics of windowing, drawing, and GUI; ws2_32, which provides a sockets interface to the networking stack; and ole32, which provides access to the Component Object Model (COM) and Object Linking and Embedding (OLE) APIs for constructing multi-component application experiences (e.g., a PowerPoint presentation with an embedded Excel chart). These DLLs correspond to libraries such as libX11 and the libraries that make up the GNOME or KDE frameworks on Unix systems.

The core of Win32 is implemented in the Windows subsystem. Divided into a kernel-mode component (win32k) and a user-mode daemon (csrss), the Windows subsystem provides roughly the analogue of an X server, a print server, and audio support. Most of the implementation resides in win32k, which implements windowing event message queues, overlapping windows, rasterization, font management and rendering, the mouse and keyboard input event queues, and a shared clipboard. csrss coordinates system initialization, shutdown, and application error reporting. Prior to Windows NT 4.0 (1996), functionality now in win32k ran in the csrss service daemon, but was moved to kernel mode to improve performance and to simplify the implementation of accelerated graphics drivers. User-mode API DLLs access win32k indirectly through stubs in user32 and gdi32; the stubs trap into a secondary system-call table of 827 functions from win32k.

In addition to the Windows subsystem, the implementation of the Win32 API depends on multiple service daemons. smss performs a role similar to Unix init, handling startup and shutdown. wininit creates read-only shared data structures, which are subsequently mapped into most processes and used by win32k to cache such common objects as default fonts, internationalization tables, and cursors. wininit also launches the components of the user's desktop upon login, the analogue of an X session manager. rpcss implements shared services for high-level inter-process communication, including COM and OLE, similar to D-Bus [24] in Linux. explorer, the Windows GUI shell, launches programs and provides shared services for drag-and-drop, "open" and "save" dialogs, and file preview. dwm implements the Windows 7 compositing window manager.

3. Approach

To maximize application compatibility while minimizing dependencies outside the library OS, we refactored Windows 7 by applying four high-level heuristics: inclusion of API DLLs based on their usage in a representative set of applications, reuse of virtualized host OS resources, resolution of dependencies through inclusion or alternative implementations, and device driver emulation. One insight we applied repeatedly in our work is the recognition that much of the code in an OS kernel and major OS subsystems is not relevant in the context of a library OS. For example, much OS code ensures security and consistency when sharing resources—either physical or virtual—between multiple applications and multiple users. In a library OS, these concerns are greatly diminished as library OS state is not shared by multiple applications or users.

Our first heuristic was to identify the API DLLs required by a representative set of applications. Those DLLs account for over 14,000 functions of the Win32 API. We used static analysis on the application binaries to roughly approximate the required set of API DLLs, and then refined the set with dynamic instrumentation by monitoring DLL load operations issued during test runs. In our experience, static analysis alone is either insufficient—DLLs can be loaded without static stubs through calls to LoadLibrary (equivalent to dlopen on Unix) with a dynamically generated string—or overly conservative—including delay-bound DLLs loaded only for specific OS versions.

Second, for kernel-mode dependencies, we implemented an NT kernel emulation layer at the bottom of the library OS. This emulation layer is quite thin, as many complex parts of a kernel—like threading, virtual memory, file system, and networking—are provided by the host OS through the security monitor. The security monitor virtualizes host resources according to a well-defined high-level ABI independent of host OS version. Other parts of the library OS are simpler because multi-user multiplexing is no longer required; for this reason the Drawbridge registry implementation is $1/50^{th}$ the lines of code of the Windows 7 equivalent. While our emulation provides most of the interfaces of the NT kernel, many calls return failure, including all requests to access or modify other processes, and almost all ioctl requests. In theory the "holes" in our emulation of the NT kernel could cause an application to fail unexpectedly due to an unimplemented corner case. In practice, we find that most applications either don't use the unimplemented interfaces, or respond gracefully when a rarely-used API returns a failure result.

Third, for dependencies on service daemons and the Windows subsystem, we either moved code into the library OS, or altered the API DLL to remove the dependency. As a rule of thumb, we included code where most of the service was relevant when running a single application, and replaced code where it was needlessly complicated by the security or consistency demands of supporting multiple applications and/or multiple users. For example, we included almost all of win32k and rpcss, as these services provide core functionality for applications. By contrast, we wrote custom library OS code to replace csrss, smss, and wininit, which primarily aid cross-application sharing of state.

Fourth, for console and human interface device dependencies, we provide emulated device drivers. We emulate the keyboard and mouse drivers required by the Windows subsystem with stub drivers that provide simple input queues, and the display driver with a stub driver that draws to an in-process frame buffer. I/O from the emulated devices is tunneled to the desktop and the user through stateless RDP connections. Our implementation of RDP reuses code from the Windows 7 kernel-mode RDP server but not the RDP device drivers, which are replaced by simpler stubs.

4. Security Monitor

The library OS interacts with the host OS through the Drawbridge ABI, which is implemented by the security monitor. The Drawbridge ABI is designed to provide a small set of functions with well-defined semantics easily supported across a wide range of host OS implementations. The ABI's design enables the host OS to expose virtualized resources to the library OS with minimal duplication of effort. We describe here first the ABI and then the implementation of the security monitor.

In providing the ABI, the security monitor enforces a set of external policies governing the host OS resources available to the application. Inspired by previous work in Singularity [33], we encode policy in manifest files associated with the application. The manifest whitelists the host OS resources that an application may access, identified by URI path. We also use the manifest as a convenient place to store per-application configuration settings.

ABI Description

The ABI includes three calls to allocate, free, and modify the permission bits on page-based virtual memory. Permissions include read, write, execute, and guard. Memory regions can be unallocated, reserved, or backed by committed memory:

```
VOID *DkVirtualMemoryAlloc(Addr, Size, AllocType, Prot);
DkVirtualMemoryFree(Addr, Size, FreeType);
DkVirtualMemoryProtect(Addr, Size, Prot);
```

The ABI supports multithreading through five calls to create, sleep, yield the scheduler quantum for, resume execution of, and terminate threads, as well as seven calls to create, signal, and block on synchronization objects:

```
DKHANDLE DkThreadCreate(Addr, Param, Flags);
DkThreadDelayExecution(Duration);
DkThreadYieldExecution();
DkThreadResume(ThreadHandle);
DkThreadExit();
DKHANDLE DkSemaphoreCreate(InitialCount, MaxCount);
DKHANDLE DkNotificationEventCreate(InitialState);
DKHANDLE DkSynchronizationEventCreate(InitialState);
DkSemaphoreRelease(SemaphoreHandle, ReleaseCount);
BOOL DkEventSet(EventHandle);
DkEventClear(EventHandle);
ULONG DkObjectsWaitAny(Count, HandleArray, Timeout);
```

The primary I/O mechanism in Drawbridge is an I/O stream. I/O streams are byte streams that may be memory-mapped or sequentially accessed. Streams are named by URIs. The stream ABI includes nine calls to open, read, write, map, unmap, truncate, flush, delete and wait for I/O streams and three calls to access metadata about an I/O stream. The ABI purposefully does not provide an ioctl call. Supported URI schemes include file:, pipe:, http:, https:, tcp:, udp:, pipe.srv:, http.srv, tcp.srv:, and udp.srv:. The latter four schemes are used to open inbound I/O streams for server applications:

```
DKHANDLE DkStreamOpen(URI, AccessMode, ShareFlags,
       CreateFlags, Options);
ULONG DkStreamRead(StreamHandle, Offset, Size, Buffer);
ULONG DkStreamWrite(StreamHandle, Offset, Size, Buffer);
DkStreamMap(StreamHandle, Addr, ProtFlags, Offset, Size);
DkStreamUnmap(Addr);
DkStreamSetLength(StreamHandle, Length);
DkStreamFlush(StreamHandle);
DkStreamDelete(StreamHandle);
DkStreamWaitForClient(StreamHandle);
DkStreamGetName(StreamHandle, Flags, Buffer, Size);
DkStreamAttributesQuery(URI, DK_STREAM_ATTRIBUTES *Attr);
DkStreamAttributesQueryByHandle(StreamHandle,
       DK_STREAM_ATTRIBUTES *Attr);
```

The ABI includes one call to create a child process and one call to terminate the running process. A child process does not inherit any objects or memory from its parent process and the parent process may not modify the execution of its children. A parent can wait for a child to exit using its handle. Parent and child may communicate through I/O streams provided by the parent to the child at creation:

```
DKHANDLE DkProcessCreate(URI, Args, DKHANDLE *FirstThread);
DkProcessExit(ExitCode);
```

Finally, the ABI includes seven assorted calls to get wall clock time, generate cryptographically-strong random bits, flush portions of instruction caches, increment and decrement the reference counts on objects shared between threads, and to coordinate threads with the security monitor during process serialization:

```
LONG64 DkSystemTimeQuery();
DkRandomBitsRead(Buffer, Size);
DkInstructionCacheFlush(Addr, Size);
DkObjectReference(Handle);
DkObjectClose(Handle);
DkObjectsCheckpoint();
DkObjectsReload();
```

We believe the brevity of the Drawbridge ABI enables tractable coding-time and run-time review of its isolation boundary. Our largest implementation of the ABI in a security monitor is 17KLoC.

We recognize that the Drawbridge ABI could be smaller. While size matters, we have in a few cases opted for exposing a slightly larger set of abstractions than was strictly necessary in order to ease porting of Windows code into the library OS. Our experience is that a slightly larger ABI (say with a dozen more calls) makes it easier to port existing code and makes the ported code easier to maintain. For example, instead of exposing semaphores, notification events, and synchronization events, we could have exposed only a single synchronization primitive. In fact, the initial version of the ABI had just 19 calls, with synchronization objects and I/O streams being coalesced into a single pipe abstraction; for example, lock release was implemented in the library OS as a one-byte send on a pipe and acquire as a one-byte write. The ABI was smaller, but reasoning about library OS implementation was more torturous. The slightly larger ABI allows the security monitor to implement virtualized resources with resources that the host OS can more efficiently support.

Implementation

The Drawbridge ABI is implemented through two components: the security monitor, dkmon, and the platform adaptation layer, dkpal. The primary job of dkmon is to virtualize host OS resources into the application while maintaining the security isolation boundary between the library OS and the host OS. Although im-

plementations of dkmon and dkpal vary across different host systems, these components are responsible for maintaining strict compatibility with the ABI specification. We currently have implementations of dkmon that run Drawbridge applications as processes on Windows 7 and Windows Server 2008 R2, on MinWin [36] built from Windows 7, and on a pre-release of the next version of Windows. We also have a version of dkmon that runs Drawbridge applications in ring 0 in a raw Hyper-V VM partition, while relying on I/O streams served from a Windows Server 2008 R2 host. Applications using the Drawbridge library OS run identically on all of these platforms.

A Drawbridge process accesses the ABI by calling dkpal. We have three implementations of dkpal: the first, which requires no changes to the host OS kernel, and uses four host OS calls to issue requests over an anonymous named pipe to dkmon; the second, which replaces the NT system-call service table on a per-process basis using techniques developed for Xax [11]; and the third, which makes Hyper-V hypercalls. dkmon services ABI requests by modifying the address space and host OS handle table of the calling process with standard Windows cross-process manipulation APIs (ReadProcessMemory, VirtualAllocEx, VirtualFreeEx, and DuplicateHandle). As an optimization, dkpal implements a few simple ABI calls (e.g., blocking wait, thread yield) by directly invoking compatible, host OS system calls; this is safe, as the Drawbridge process cannot create host OS handles. dkpal currently calls 15 distinct host OS system calls. Eventually, we expect to move the data paths dkmon into the host kernel to avoid the cost of a complete address space change to service some ABI calls, and to harden the boundary around Drawbridge processes; this will require an update to dkpal, but no changes in the Drawbridge library OS.

dkmon uses host NT threads and synchronization objects—semaphores, notification events, and synchronization events—to implement the scheduling objects exposed through the ABI. As a result, Drawbridge threads reside in the host kernel's scheduling queues and avoid unnecessary scheduling overheads.

I/O streams are used by the library OS to implement such higher-level abstractions as files, sockets, and pipes. The security monitor filters access to I/O streams by URI based on a manifest policy; where access is allowed, it directs I/O to the mapped resources. This indirection enables run-time configuration of the application's virtual environment, and prevents applications from inadvertently or maliciously accessing protected resources within the host system's file namespace. Unless overridden, the monitor's default policy only allows a Drawbridge process to access files within the same host directory as its application image.

As an example, our library OS leverages I/O streams to emulate NT file objects and named pipe objects, as well as a proxied interface to networking sockets. In the latter case, the library OS includes a minimal version of ws2_32 that use I/O streams identified by tcp: and udp: URIs. The security monitor backs these streams with sockets provided by the host system's ws2_32. The current approach was taken as an implementation expediency; Drawbridge could provide better isolation by offering an IP-packet stream interface and moving the implementation of TCP and UDP into the library OS.

5. Windows Library OS

Refactoring Windows 7 into a library OS constituted the largest portion of the work to realize Drawbridge. The main challenges included: orchestrating process bootstrap with minimal changes to existing components, emulating host kernel interfaces in user mode on the Drawbridge ABI, porting system-wide application services to run in-process, and enabling process serialization and deserialization. We believe that these challenges would be typical of the refactoring effort required to convert any complex, modern operating system into a library OS for desktop applications, and that our solutions would apply elsewhere. However, we also recognize that our task was made significantly easier because most Windows applications seldom use child processes and the Windows API has no fork primitive.

OS Library Bootstrap

Bootstrapping is a challenging task in any OS, balancing the demands of efficiency and good engineering. For example, should one first initialize the lock manager, which requires memory for storage, or the memory manager, which requires locks for synchronization? Some of these complexities remain for the library OS, which aggregates per-process code and state with formerly system-wide OS code and state.

Process Bootstrap To create a Drawbridge process, dkmon uses the host OS's native facilities to create a suspended process containing a bootstrap loader (dkinit). Every NT process is created with the ntdll library mapped copy-on-write, because the kernel uses fixed offsets in the library as up-call entry points for exceptions. Before allowing a new process to execute, dkmon maps its own dkntdll library into the new process's address space and overwrites upcall entry points in the host-provided ntdll with jumps to dkntdll, eviscerating ntdll to a jump table and replacing it as the dynamic loader. dkmon writes a parameter block into the new process's address space to communicate initialization parameters, such as a reference to the pipe to be used for communication with dkmon. dkmon then resumes the suspended process, with execution starting in ntdll and immediately jumping to dkntdll, which sets up initial library linkage (to itself) and transfers control to dkinit. dkinit invokes dkntdll to initialize the win32k library (described next) and to load the application binary and its imported libraries. When loading is complete, dkinit jumps to the application's entry point.

Win32k Bootstrap Converting win32k from a kernel subsystem to a user-mode library required reformulating its complicated, multi-process initialization sequence. In standard Windows, first, the single, system-wide instance of win32k is initialized in kernel mode. Second, wininit initiates the preloading of win32k's caches with shared public objects such as fonts and bitmaps. Because win32k makes upcalls to the user32 and gdi32 user-mode libraries to load an object into its cache, these libraries must be loaded before filling the cache. Third, when a normal user process starts, it loads its own copies of user32 and gdi32, which connect to win32k and provide GUI services.

We considered refactoring the initialization of win32k, but rejected that idea as it required a deeper fork of the source code than desired and prevented sharing of code between full Windows and Drawbridge. Instead, we opted for an approach that simulates the full win32k bootstrapping sequence within a single process. We exported entry points from win32k, user32, and gdi32, which are called by dkinit for each of the boot steps.

Much of the effort described here can be viewed as unwinding the complexity of the multi-server architecture of Windows into an in-process library OS. On a full Windows system, csrss creates a read-only, shared-memory segment to share cached bitmaps and fonts, which is replaced in the library OS with heap allocated objects, since all components that access it now share the same address space and protection domain. The upcalls that torture the

win32k initialization sequence were needed only because a shared, trusted win32k must avoid being confused into loading one principal's objects on behalf of another; in Drawbridge, that responsibility is removed, and the corresponding complexity can be reduced. Likewise, we removed many other access checks over shared state from win32k, and eliminated the Windows logon session abstraction, wininit, and csrss.

Emulating NT Kernel Interfaces

To support binary compatibility with existing Windows 7 API DLLs, we provide user-mode implementations of approximately 150 NT kernel system calls. The majority of these functions are stubs that either trivially wrap the Drawbridge ABI (e.g., virtual memory allocation, thread creation, etc.), return static data, or always return an error (i.e., STATUS_NOT_IMPLEMENTED).

The remaining system calls produce higher-level NT abstractions, such as files, locks, and timers, built entirely inside the library OS using Drawbridge primitives. The most challenging part of this work was building compatible semantics for the NT I/O model, including synchronous and asynchronous I/O, "waitable" file handles, completion ports, and asynchronous procedure calls.

Additional NT emulation calls were required to win32k from kernel mode to user mode. The interfaces provided to kernel-mode libraries by the NT kernel are largely disjoint from those provided to user-mode libraries; we added roughly 6.3KLoC to emulate these kernel-mode calls.

Shared System Services

Windows applications depend on shared system services, accessible either through system calls or IPCs to trusted service daemons. These include services such as the Windows registry and OLE. In order to confine the state and dependencies of Drawbridge processes, our library OS implements the functionality of several such services using two design patterns: providing simple alternate implementations of service functionality, and hosting extant library code in-process.

Alternative Implementations Many Windows system services are backed by complex and robust implementations, tuned and hardened for a wide variety of use cases. For a few such services like the registry, it proved more practical to reimplement the services' advertised interfaces within the library OS rather than port their existing implementations.

The *registry* is a system-wide, hierarchical, key-value store, accessible to Windows processes through system calls. The traditional Windows registry is implemented in kernel mode with 61 KLoC; its complexity is required to implement fine-grained access control and locking as well as transactional semantics. While we might have replicated code for the kernel registry into the library OS, we found that registry code is inseparably connected to code for the kernel object namespace and the kernel scheduler. The Drawbridge library OS instead includes a private, in-process reimplementation of the registry, significantly simpler than the shared kernel registry. Drawbridge's NT emulation layer supplies a simple 1.3 KLoC interface to this implementation, with coarse locking and no support for transactions.

Importing Implementations In several cases, we encountered shared Windows services whose implementations could be ported largely intact. As an example, Drawbridge's support for COM required functionality traditionally provided by rpcss. Supporting COM is an absolute requirement for rich applications in Windows; a significant number of desktop- and server-class applications are formed by composing multiple COM objects through the OLE protocol. With one exception (Reversi) each of our test applications uses OLE.

Refactoring COM followed the same basic pattern used with other system services: shared, out-of-process components were ported to run privately in-process and bound directly to application-side libraries. For example, a key component of OLE is the running object table (ROT), which provides inter- and intra-process name resolution for COM objects. While only one instance of the ROT is maintained per system within rpcss, in Drawbridge it runs directly within the application process and only manages local objects. Fewer than 500 lines out of 318 KLoC were changed in the COM runtime (ole32) to make these changes.

Process Serialization

The volatile state associated with a traditional Windows process is distributed across the NT kernel and kernel-mode drivers; shared, out-of-process, user-mode service daemons; and the process's own address space. Serializing the running state of a Windows process would require the careful cooperation of each of these components, and significant changes to the OS. However, with Drawbridge process isolation, a process's transient state is either confined to pages in its address space or can be reconstructed from data maintained within its address space.

Due to careful design of the ABI and library OS, serializing a Drawbridge process is relatively simple. Running win32k as a user-mode library vastly reduced the amount of kernel state associated with the process. The remaining out-of-process state consists of resources managed by the host OS, such as files and synchronization objects. To account for this, our implementation of the ABI inserts indirection between Drawbridge system objects and host NT system objects. This distinction enables the library OS to easily unbind and rebind these objects at deserialization time. Because the metadata for host kernel objects are stored within a process's address space, serialization only requires quiescing the threads and serializing the contents of the address space. The thread contexts need not be serialized, as the active register contents are stored on their stacks during quiescence. The application serializes itself with no involvement from the host, beyond the I/O stream to which the serialized state is saved.

In order to quiesce the threads within a Drawbridge process, the security monitor signals a notification event to indicate a serialization request. Threads blocked on Drawbridge ABIs are woken via the notification event, outstanding I/O requests are completed, and other threads are interrupted with an exception. Once notified of the pending serialization, all threads, except one, yield indefinitely. The final thread begins serializing the process to a monitor-provided I/O stream, recording virtual memory bookkeeping information and the contents of the process's address space. Files on which the process depends must be migrated with the application or accessed through a distributed file system. Network sockets are terminated on migration causing applications to reestablish their network connections. In a world of migrating laptops, we find that applications are robust to network interruptions. After serialization is complete, the yielding threads are woken and the process continues normal execution.

Reconstructing a Drawbridge process from its serialized state requires adjustments to its initialization sequence. Instead of loading the application binary, serialized virtual memory data are used to restore the contents of the process's address space, including heap and stack memory as well as memory-mapped I/O streams. Code in dkpal rebinds host system objects to ABI objects without

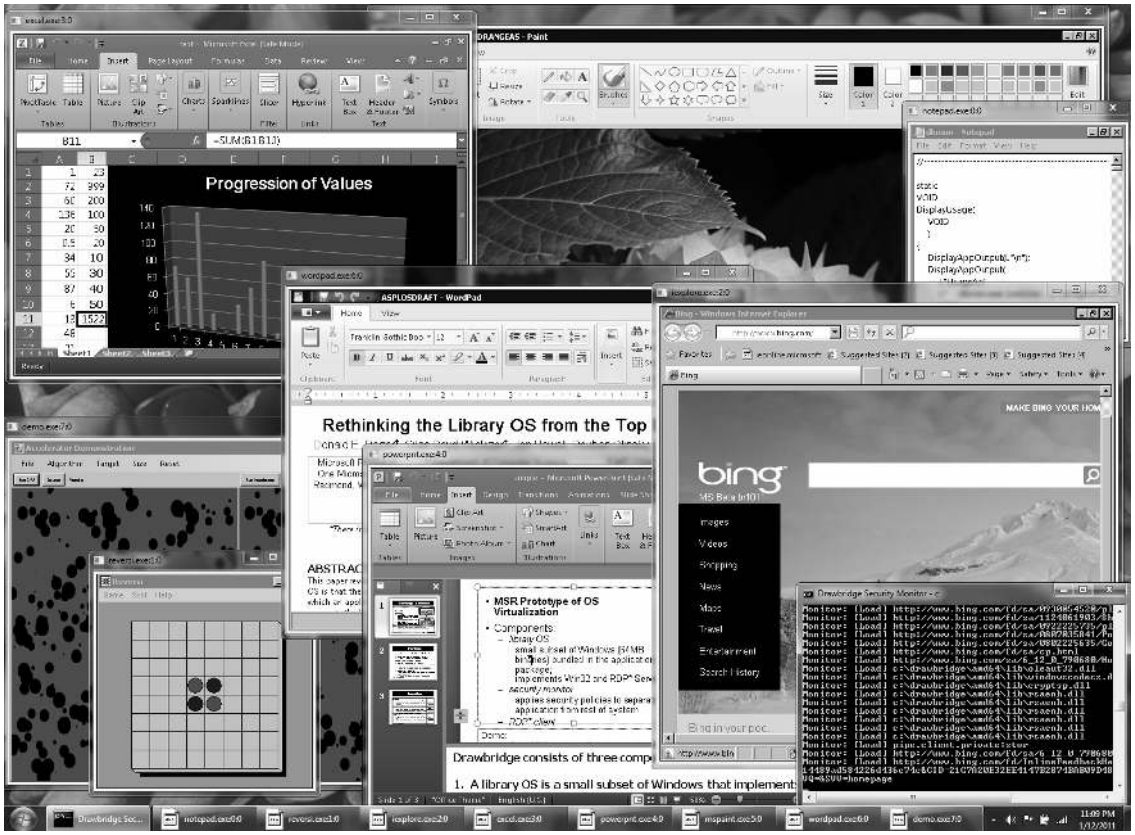

Figure 3. Screenshot of Drawbridge Applications: Clockwise from the top-left: Excel, Paint, Notepad, dkmon, Internet Explorer, WordPad, PowerPoint, Reversi, and a CLR Demo.

involvement from the library OS or the application. After recreating the process, the threads quiesced during serialization are unblocked and continue normal execution.

Limitations

The present Drawbridge system is a research prototype; it is far from a production system. While Drawbridge supports over 14,000 Win32 API functions, this is a fraction of the total Win32 API. At the time of writing, Microsoft has no plans to productize any of the concepts prototyped in Drawbridge.

The two holes in the current implementation are support for printing and support for multi-process applications that communicate through shared state. The challenge of printing is that most Windows printer drivers actually load in-process with the application. As our design requires leaving hardware drivers within the host OS, they can't be loaded into the library OS without creating undesirable dependencies. A possible solution is to reuse the approach of RDP 7.0, which employs a universal application printer driver that prints to a common format, XPS, and then asks the RDP client's printer to print the XPS.

Solving the problem of multi-process applications is much harder, particularly for applications that communicate through shared state in win32k, as is done in many OLE scenarios. For example, Microsoft Outlook can be configured to use Microsoft Word as a text editor with the shared state passing through win32k message queues. We have considered, but not implemented, two possible designs. One is to load multiple applications into a single address space. Another is to run win32k in a separate user-mode server process that can be shared by multiple applications in the same isolation container. The latter approach follows a pattern commonly used by microkernel designs.

A third weakness in the present implementation is that code paths in many Win32 APIs ultimately lead to NT APIs that are not implemented by our emulation layer. We have ported the most frequently used subset of a very large API space. As we have run new applications, the number of additions to the library OS has diminished, but not completely disappeared. For example, after Excel ran, getting PowerPoint to run on Drawbridge took only two days: one to fix a bug we had introduced in win32k, and one to implement an additional API in the NT emulation layer.

Finally, there are classes of applications that, by design, will probably never run in Drawbridge. For example, administration tools, that manipulate the host OS, or development tools such as debuggers, which need unfiltered access to the environment, cannot be run without creating hard dependencies on the host OS.

6. Experiments

This section evaluates the theses of the Drawbridge project: that a library OS can run rich desktop applications, that such refactoring is feasible, that it is suitable for isolating large numbers of applications in a single computer, that it protects the integrity of the host OS at least as well as a VMM, that it provides greater mobility for running applications, and that it enables independent evolution of host OS from library OS. We also measure the servicing implications of the library OS.

All data were collected on an HP z800 Workstation with dual 2.4GHz Intel Xeon E5530 Quad-Core CPUs with hyper-threads disabled, 16GB of RAM, and dual 10,000 RPM hard drives. All Windows experiments use 64-bit Windows 7, Ultimate Edition, with the page file disabled. All Hyper-V experiments run on 64-bit Windows Server 2008 R2, Enterprise Edition, which shares the same code base as Windows 7. Drawbridge and all applications are 64-bit binaries. Windows 7 and Hyper-V were tuned for maximum scalability according to published best practices [27].

For most experiments, we present results for three applications: Excel, a canonical desktop application; Internet Explorer, a web browser and network client application; and IIS, a canonical server application. Unless stated otherwise: Excel experiments used small (11KB), large (20MB), and huge (100MB) spreadsheets; Internet Explorer rendered research.microsoft.com, and IIS was serving up the default Visual Studio 2010 ASP.NET application using CLR 4.0.

Running Applications

The primary hypothesis of Drawbridge is that a legacy OS can be refactored into a resource-efficient library OS that supports a large class of rich, desktop applications. Figure 3 shows a sample screenshot with output from dkmon and seven applications running in Drawbridge: Microsoft Excel 2010, Windows 7 Paint, Notepad, Internet Explorer 8, PowerPoint 2010, Windows 7 WordPad, a CLR demo application, and Reversi, the first game to run on Windows. Other applications we have tested include IIS 7.5, DirectX 11 demos, and a number of in-house applications. With the exception of disabling software licensing in Excel and PowerPoint, all application binaries are unmodified.

To run on Drawbridge, applications were "sequenced", by running their setup programs on a desktop copy of Windows 7, and capturing the file-system and registry changes made by the setup program into a Drawbridge package. Tools for collecting file-system and registry changes are well understood and deployed with products such as Microsoft Application Virtualization (App-V) and VMWare ThinApp [37].

Cost of Refactoring

Our hypothesis maintains that the refactoring task is tractable. Of the 93 binaries containing executable code in the Drawbridge library OS, 62 come directly from Windows 7 with no modifications, 12 are alternative implementations (mostly machine generated stubs), and 19 contain modifications. The changes are generally quite small when compared with the number of lines of code that remain unchanged (see Table 1). The most significant changes were in: gdi32, ntdll, and user32, which no longer trap, but now directly call either win32k or the NT emulation layer; dxapi, dxg, and win32k, which were kernel-mode drivers now modified to run as user-mode DLLs; and ole32 and wininet, in which we removed dependencies on external services. Supporting our hypothesis, repurposing 5.6 MLoC of Windows 7 into a library OS required less than 16 KLoC of changes (0.3% of code base) and 36 KLoC of new code; the entire project was completed in fewer than two person-years.

Binary	#if's	Changed LoC	Total LoC	Size (KB)
advapi32.dll	18	720	61,975	655
dxapi.dll	88	294	3,225	18
dxg.dll	110	449	12,706	85
dxgi.dll	4	20	46,347	616
dxgkrnl.dll	1	8	518	9
gdi32.dll	9	70	43,711	374
kernel32.dll	37	262	153,905	944
kernelbase.dll	21	188	40,734	312
msvcrt.dll	1	5	70,201	590
ntdll.dll	83	4,274	148,327	1,484
ole32.dll	188	915	196,706	1,907
oleaut32.dll	5	34	84,331	763
rdp4vs.dll	5	647	50,312	261
rdpclip.dll	18	111	21,306	128
rdpvdd.dll	1	666	4,312	30
rpcrt4.dll	5	34	135,812	959
user32.dll	68	497	60,161	935
win32k.dll	685	5,341	343,082	2,845
winhttp.dll	15	1,146	6,225	40
New Implementations: clbcatq.dll, ddraw.dll, dwmapi.dll, iphlpapi.dll, msi.dll, netapi32.dll, sechost.dll, secur32.dll, wininet.dll, winspool.drv, ws2_32.dll, wtsapi32.dll			25,984	251
Unchanged: atl.dll, comctl32.dll, comdlg32.dll, comsvcs.dll, crypt32.dll, cryptbase.dll, cryptsp.dll, d3d10.dll, d3d10_1.dll, d3d10_1core.dll, d3d10core.dll, d3d10level9.dll, d3d10ref.dll, d3d10sdklayers.dll, d3d10warp.dll, d3d11.dll, d3d11ref.dll, d3d11sdklayers.dll, d3d8thk.dll, d3d9.dll, d3dcompiler_42.dll, d3dx10_42.dll, d3dx11_42.dll, d3dx9_42.dll, dbghelp.dll, ddrawex.dll, dui70.dll, duser.dll, explorerframe.dll, fms.dll, gdiplus.dll, iertutil.dll, imagehlp.dll, mfc42u.dll, mlang.dll, msasn1.dll, msftedit.dll, msls31.dll, mswsock.dll, odbc32.dll, oleacc.dll, profapi.dll, propsys.dll, psapi.dll, rdpd3d.dll, rgb9rast.dll, rsaenh.dll, shell32.dll, shfolder.dll, shlwapi.dll, sspicli.dll, uiribbon.dll, urlmon.dll, userenv.dll, usp10.dll, uxtheme.dll, version.dll, windowscodecs.dll, winmm.dll, wintrust.dll, wsock32.dll, xmllite.dll			3,995,244	57,912
Totals	**1,362**	**15,681**	**5,505,124**	**71,118**

Table 1. Summary of changes to Windows 7 executable binaries to produce the Drawbridge library OS.

Overheads

To validate the hypothesis that Drawbridge has modest resource overheads, we measure committed memory and start times for

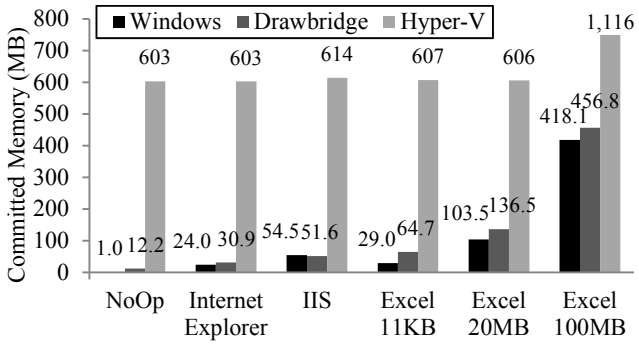

Figure 4. Memory per application (including memory used by library OS or guest OS).

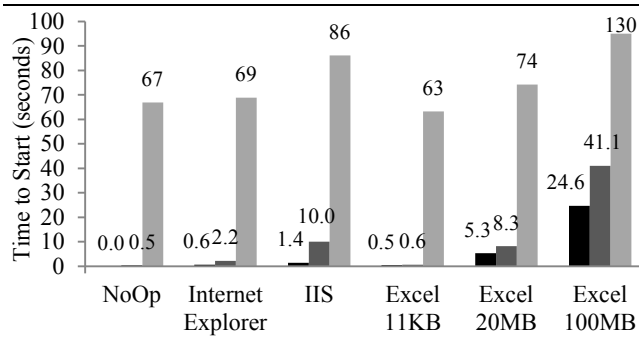

Figure 5. Time to start application (including time to start library OS or guest OS).

applications running natively on Windows, on Drawbridge, and on Hyper-V. Figure 4 shows the amount of committed memory required for each OS configuration and application. All VMs were configured with 512MB of RAM, except Excel 100MB test, which has 1024MB to avoid excessive paging. With 512MB, the startup time is over 260 seconds. The additional committed memory of Drawbridge is basically the cost of each application running a private copy of win32k, including its fonts and graphic object caches. Running the application in a VMM, on the other hand, incurs the full cost of running a complete guest OS. Similarly, startup times for a full guest OS are much higher than for a library OS (see Figure 5). Ongoing execution overheads are only slightly higher with Hyper-V (for our applications typically less than 1%).

Figure 6 shows the aggregate effect of memory for Excel. While Hyper-V can host only 23 isolated copies of Excel (each with a 20MB spreadsheet loaded), Drawbridge can host 104 instances on the same hardware, compared to 142 instances if Excel is run as a native Windows application with no isolation. Drawbridge is sufficiently efficient that every application can be run in its own isolation container. For Internet Explorer, we found that Drawbridge can host 527 instances, Windows can host 138, and Hyper-V can host 22 (see Figure 7). In the case of Windows, Internet Explorer exhausts the hard limit on GDI handles per logon session in win32k, not physical memory. Drawbridge does not have this limit as each library OS has its own private win32k. Finally, for IIS, we found that Drawbridge can host 287 instances, Windows can host 266 and Hyper-V can host 21 (see Figure 8).

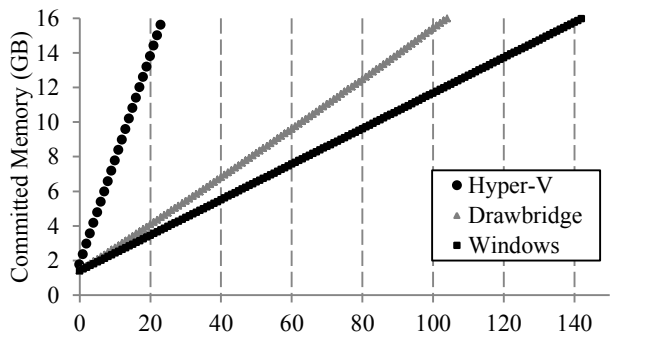

Figure 6. Memory committed for increasing copies of Excel, each with a 20MB spreadsheet loaded.

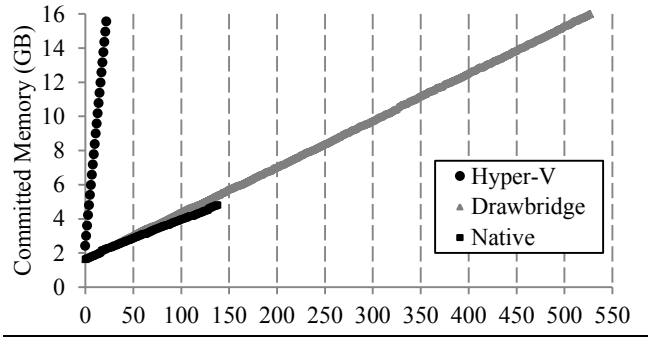

Figure 7. Memory committed for increasing copies of Internet Explorer rendering research.microsoft.com.

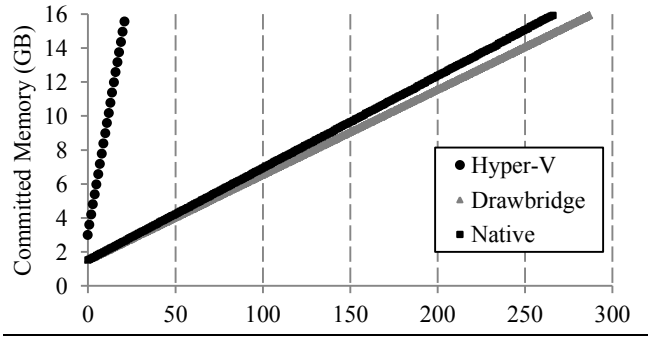

Figure 8. Memory committed for increasing copies of IIS.

In terms of disk footprint, the Drawbridge library OS is 64MB (or 83MB with DirectX 11 support) compared to 4.2GB for a guest OS copy of Windows 7. If desired, the library OS can be further reduced to meet the needs of a single application; a copy of the library OS reduced to its minimum for running Reversi is under 16MB.

Protecting System Integrity

To illustrate that Drawbridge applications are isolated and cannot harm the host OS, we performed two simple experiments and conducted a case study based on a recently-published Internet Explorer exploit [20]. First, we ran a toy malware program that

deletes every registry key, which would normally leave a system unusable, and found that only the malware was affected. Second, we wrote a simple key logger using the SetWindowsHookEx function. When run outside of Drawbridge, the key logger collected every keystroke issued. Inside Drawbridge, the key logger was unable to obtain keystrokes in other applications.

Keetch [20] documents a set of attack vectors and attack patterns to exploit possible weaknesses in the protected mode of Internet Explorer. Protected mode runs a copy of Internet Explorer in a process with restricted OS permissions. Among other restrictions, in protected mode, Internet Explorer should not be able to write to persistent storage. The idea of protected mode is that even if an attacker is able to run exploit code within Internet Explorer, the code cannot escape the low privilege process and therefore no permanent harm can be done to the system. Keetch identifies five attack vectors for exploit code to escape protected mode: name squatting on the NT kernel object namespace, leaked or duplicated handles, objects deliberately shared between low- and fully-privileged processes, clipboard content spoofing, and spoofing of a web server in the local-domain trusted zone through the loopback interface.

Drawbridge mitigates all five of the attack vectors identified by Keetch. Each application runs with its own library OS that doesn't share an emulated kernel namespace, doesn't share handles, and doesn't share kernel objects. Drawbridge applications can access the clipboard and act as network servers, but only if the ability has been whitelisted for the application. Internet Explorer on Drawbridge is not permitted to act as a web server, thus preventing network spoofing. Access to the clipboard in Drawbridge is set by RDP policy and can also be prevented. By comparison, Windows 7 running in a VMM doesn't share its kernel namespace, doesn't share handles, and doesn't share kernel objects. Furthermore, like Drawbridge, a VMM guest OS can access the desktop clipboard and the local network only if permitted.

Migrating Applications

One benefit of Drawbridge's ABI is that it enables the migration of running applications between computers and across OS reboots. For most Windows applications, the only alternative to Drawbridge is to use a VMM to migrate the application and its full operating system. To compare the cost of migration of an application using Drawbridge versus a Windows 7 guest VM using Hyper-V, we ran five application scenarios on each system, triggered a memory snapshot, and then compressed the snapshot to determine the smallest image. We disabled the page file in Windows 7 on Hyper-V to avoid including the additional size of a page file in the snapshot.

As Table 2 shows, Drawbridge snapshots are significantly smaller, because it doesn't serialize the OS with the application. In practice, we find that application snapshots for Drawbridge are typically in the 3-4MB range; roughly the size of an MP3, they are easily moved over network connections. Application serialization and deserialization generally takes less than 1 second on Drawbridge, and more than 10 seconds for Hyper-V.

Evolving the OS Kernel

By factoring the OS into three components (high-level user shell, library OS, and low-level kernel), Drawbridge enables their independent evolution. To validate this assertion we wrote a version of dkpal designed to run as a Hyper-V partition in ring-0, replacing the low-level Windows 7 kernel implementation.

Application	Drawbridge Snapshot	Hyper-V Snapshot
Excel: 11KB file opened	3.1 MB	148.6 MB
Excel: 20MB file opened	21.2 MB	155.8 MB
Excel: 100MB file opened	86.9 MB	313.2 MB
Internet Explorer	3.9 MB	184.8 MB
IIS: no site pre-loaded	1.1 MB	193.4 MB

Table 2. Comparison of compressed memory snapshots.

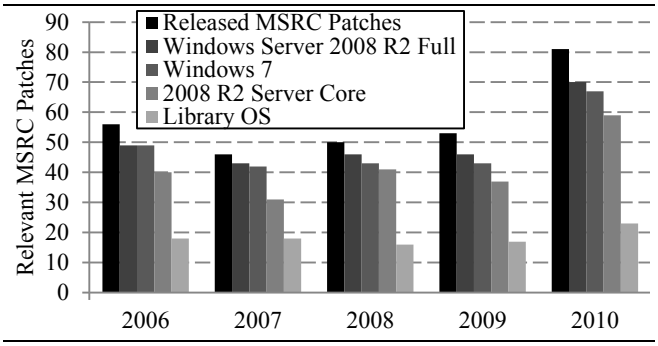

Figure 9. Security patches affecting binaries in Windows Server 2008 R2, Windows 7, Windows Server 2008 R2 Server Core, and Drawbridge.

hvdkpal is essentially a small OS kernel that implements the Drawbridge ABI described in Section 4. hvdkpal implements threading and preemptive scheduling by multiplexing Hyper-V virtual CPUs. hvdkpal includes an implementation of futexes [14], which it uses to provide Drawbridge events and semaphores. Since hvdkpal runs in ring-0, it is able to manipulate page tables directly in order to manage virtual memory. hvdkpal does not implement a file system or networking stack; to support I/O streams, it forwards I/O-related ABI calls to the host Windows Server 2008 R2 kernel running in Hyper-V's root partition. The hvdkpal implementation is about 6,000 lines of C.

Despite the large differences between the hvdkpal kernel and the Windows kernel applications expect, hvdkpal runs our entire suite of unmodified Windows applications with an unmodified library OS and an unmodified high-level user shell. The applications are oblivious to the fact that they are running in ring-0.

We also ran our set of unmodified applications and library OS on a MinWin [36] build of Windows. MinWin is a minimal build of Windows with less than 48MB of binaries; it consists of the NT kernel, the NTFS file system, TCP/IP stack, and the storage and network drivers for a specific computer. It does not include a shell. For our experiments, we connected the RDP client directly to a graphics frame buffer driver. One of our colleagues has begun to experiment with Windows 7 applications on the Barrelfish OS [6] using the Drawbridge ABI to explore extreme hardware configurations. Our experiments indicate that Drawbridge's approach to factoring OS components enables independent evolution of the low-level kernel from the library OS and applications.

Servicing the Library OS

Once a month, on "Patch Tuesday", Microsoft issues security bulletins (MSRCs) describing software security vulnerabilities and releases patches to those vulnerabilities. The amount of code inside the library OS is smaller than the code in the full OS, so one would expect fewer patches to affect the library OS. This

intuition is validated by an analysis of which binaries were affected by each security-related patch from 2006 to 2010.

As Figure 9 shows, the library OS requires significantly fewer security updates due to its smaller code size. Compared with Windows 7, Windows Server 2008 R2, and Windows Service 2008 R2 Server Core, the Drawbridge library OS has a much smaller servicing footprint. Of 286 security bulletins, 254 have resulted in patches to OS computers. Just 36% of patches, 92, affect the library OS; the rest affect code found only in the host OS. Furthermore, some of the patches in the library OS might be redundant if they prevent one application from compromising another. The comparison with Server Core installation of Windows Server 2008 R2 is interesting because Server Core is positioned as a way to reduce the attack surface of servers by removing "client" components such as the GUI shell. Drawbridge is affected by roughly half as many MSRC issues as Server Core over the same time period.

7. Related work

This section surveys related work, primarily: previous library OSes, which adopt similar designs in service of different goals, and virtual machines, which provide many similar benefits at a higher cost. We briefly discuss other approaches to achieving strongly isolated processes.

Previous Library Operating Systems

Anderson [3] initially proposed that larger portions of the OS be factored into application libraries, primarily to improve performance. A general-purpose OS often applies resource management heuristics that can work at cross-purposes with a particular application. For instance, file read-ahead is a common optimization that is useful in the common case (sequential file access), but can harm performance of an application that accesses its data randomly. This idea was brought to fruition in the Cache kernel [10], Nemesis [21], and Exokernel [13]. The Cache kernel design focused on allowing applications fine-grained control over thread scheduling and memory swapping. The Nemesis OS was particularly focused on eliminating scheduling and performance interference through shared resources, in order for multimedia applications to meet real-time deadlines. The Exokernel design is closest to the bare hardware interface of a modern VMM.

Although previous library OS systems focused on performance, rather than encapsulation of the OS personality, Drawbridge shares with them several design points. At a high-level, the Drawbridge design is similar to the original Exokernel design, which had a kernel responsible for controlling access to the low-level hardware, and the rest of the functionality in the application library. The interface between the Exokernel and library OS was much closer to the hardware interface of a modern virtual machine, whereas Drawbridge provides higher-level APIs of threads and virtual memory, similar to the Cache or Nemesis kernels. The Exokernel also allowed applications to load constrained (or "safe") extensions into the kernel for management of hardware resources that could not be reasonably exported into a non-privileged address space. For instance, Exokernel library OSes would install packet filters into the lower-level kernel to multiplex the network. In contrast, both Drawbridge and the Cache kernel delegate low-level resource management to the kernel and neither system permits a library OS to load code into the kernel.

Relative to previous library OS designs, the first contribution of our work is to show that a large commercial OS can in fact be refactored into a library OS; this is in contrast to the minimal library OS implementations of previous systems. Second, Drawbridge incorporates an ABI design that streamlines the library OS implementation without compromising security isolation or unduly exposing implementation details of the underlying host OS. As a proof of this point, Drawbridge is the first library OS system of which we are aware that supports either process serialization or migration of the guest application across diverse host systems. Finally, the Drawbridge ABI presents higher-level abstractions that trivially share host OS resources, such as the CPU and buffer caches. We believe the Drawbridge ABI offers a better starting point for library OS construction.

Virtual Machines

Virtual machines [35] are the primary success story for packaging an application and its dependencies on a fully-featured OS into a single self-contained unit. Research systems, such as The Collective [31], use a single virtual machine per application to encapsulate an application and its OS. Treating a complete legacy operating system, including the kernel, as a library for a single application wastes substantial memory and computation. This waste comes from an "impedance mismatch" that causes lost sharing opportunities, lost statistical multiplexing opportunities, and allocation strategies designed for physical resources running against virtual ones. Only through significant research and development effort has this waste been reduced.

New *paravirtualization* approaches, which warp the VM interface gradually farther away from a raw hardware interface, expose instead resources that a guest operating system can use more efficiently [5, 12, 17, 38, 39]. Despite significant research effort, however, running a full legacy operating system in a VM still incurs substantial overhead, as the legacy operating system brings with it system management processes and large allocations of kernel memory. Each guest can demand gigabytes of disk storage and hundreds of megabytes of physical RAM (perhaps reduced by clever sharing). Ultimately, paravirtualization is only chipping away at the margins; paravirtualized VMs will not scale to the same level as OS processes without more drastic measures.

A key contribution of Drawbridge is judicious selection of extremely paravirtualized VM abstractions in the Drawbridge ABI. This ABI demonstrably alleviates the overheads of hardware virtualization without unduly constraining OS design or compromising isolation. Roscoe *et al* [30] argued for research into higher-level hypervisor abstractions, Drawbridge answers that call.

Other Approaches to Library OS Goals

OS customization Like previous library OSes, Libra [2] allows applications to customize their OS within a virtual machine in order to improve application performance. In Libra, an application that needs to extend the OS runs in a separate VM. The default OS functionality is provided by a sibling virtual machine over a shared memory protocol, and applications can selectively service their own requests from a custom implementation.

The performance concerns of Libra and other library OSes are largely complementary to Drawbridge; one could imagine customizing versions of the Drawbridge library OS with application-specific performance optimizations. More than optimizing performance, Drawbridge is concerned with encapsulating the personality of a library OS and enabling applications to run on future host OSes.

Xax and NaCl Recent research on isolating untrusted web applications demonstrated that a limited set of OS abstractions were sufficient for porting large libraries of legacy code, albeit not

entire interactive GUI applications [11]. The isolation technique proposed in Xax, namely hardware memory protection and a limited system call table, is shared by Drawbridge; NaCl [40] uses alternate techniques based on software fault isolation.

Stronger isolation in monolithic kernels The monolithic organization of conventional operating systems makes them difficult to secure. In order to avoid the loss of functionality and performance, a number of systems have attempted to augment monolithic systems with additional security checks in a minimally disruptive manner. For instance, the Linux VServer [32] adds additional access control list checks and imposes a separate namespace for kernel objects used by an isolated process. All kernel data, however, reside within a shared, monolithic kernel address space. This approach of adding security checks and/or replicated kernel data structures is also exemplified by SELinux [23], zones [29], jails [34] and containers [7], however it suffers several drawbacks.

First, it is difficult to know that the job of inserting additional checks is done, or that the set of hooks is complete in a complex and rapidly growing kernel. Second, the abstractions to which the hooks must be added are exactly those that were not designed to express restrictions; therefore, the resulting mechanisms may not lend themselves to a reasonable policy calculus. For instance, a simple policy such as "Do not allow the contents of file X to leave the machine" must be translated into a perilous series of decisions on whether to allow innocuous-looking accesses to local system resources [15, 41].

It is tempting to try to paint a layer of isolation functionality onto an existing monolithic system, but the result is hard to trust since it requires maintaining properties across a large interface to a large and evolving code base. Often, the structure of a monolithic system makes it difficult even to rigorously specify isolation. Therefore, Drawbridge avoids a monolithic organization in favor of a library OS design, which lifts much OS functionality into the application's strongly isolated address space.

We hypothesize that library OSes will prove a more amenable platform for strong security isolation than a monolithic kernel, with a potentially simpler mapping of policies onto concrete decisions that must be made in the security monitor. Previous systems have explored policies much more thoroughly than this work, and we defer their design to future work.

Compatibility through API emulation Several research systems have attempted to provide compatibility with legacy OSes by emulating their APIs [4, 16, 18]. Although emulation can work for common, heavily used functions, it is highly prone to subtle inconsistencies in the less used functions. Emulating the Windows API on Unix has required a colossal effort and consistently remained several releases behind Windows in terms of compatibility [30]. Even among Unix implementations, calls such as `setuid` are prone to compatibility issues [9]. In terms of compatibility, there is simply no substitute for bundling an application with the appropriate components of the original OS.

Application Virtualization The need to robustly package applications without the high cost of VMM solutions has driven development of at least two commercial products. Both App-V [26] and ThinApp [37] virtualize calls to the file system and registry. By doing so, they allow users to run desktop application like Microsoft Office without running a setup program. Because they virtualize only a subset of the file system and registry, they are easily circumvented or broken by new OS releases, and provide no migration for live applications. However, the application "sequencing" techniques used by these systems to create application packages are complementary to Drawbridge.

8. Discussion

The library OS allows a new model for packaging applications and new opportunities to reshape the computing experience. The fundamental problem of application packaging is deciding how to separate an application and its dependencies from their environment so the application can run predictably across a nearly infinite variety of user deployments. An ideal packaging technology makes it easy to deploy an application, continue the application across changes such as an OS upgrade, relocate the application from one computer to another, prevent the application from corrupting its host, and prevent the host from corrupting the application.

The library OS offers a new way to package applications. Applications can either be packaged with their dependent library OS, or can contain metadata informing the host OS of their required library OS. With the latter approach, a vendor could distribute and service a number of library OS images—say Windows XP, Windows Vista, and Windows 7 library OSes—with a single host OS.

For researchers and developers, the library OS approach offers the ability to rapidly evolve OS kernels, shells, and APIs independently without breaking other OS components or applications. Easily serialized applications offer opportunities to improve debugging and testing through techniques such as time-travel debugging, easy cloning of running application instances, and easy release of experimental OS components. For example, colleagues at Microsoft Research have used Drawbridge to deploy prototypes of new Win32 APIs that enable distributed user experiences.

Application Compatibility

Maintaining compatibility with application binaries across multiple OS releases is a significant challenge. While OS vendors expend significant resources maintaining compatibility, each OS release inevitably results in a large number of broken applications. Techniques for maintaining application compatibility in Windows 7 include one-off code in APIs; application compatibility shims, which load as intermediate code between application and API to alter input or output parameters; and Windows XP mode, which is a complete VM image of the latest release of Windows XP.

Application compatibility is particularly onerous for enterprises where one-off line-of-business (LOB) applications are abundant. The recourses available to an enterprise when a LOB application is broken by a new OS release are often limited as programming staff has moved to other projects and source code may be missing.

By encapsulating the portions of the OS most likely to break application compatibility, the library OS offers a new technique to resolve application compatibility problems. Each application could run with the library OS of its choice. Programmers or IT administrators could bind an application to a known good library OS, which would continue to run correctly on top of new host OS releases. For example, an application packaged with a Windows XP library OS could continue to run correctly on Windows XP, Windows Vista, or Windows 7 host OSes. Conversely, an update to Windows 7 could ship with library OSes for Windows XP and Windows Vista. Users could upgrade to new OS releases unconstrained by the complex calculus of application compatibility.

Applications might even run on down-level OS releases. For example, an application with the Windows 7 library OS could run on Windows XP. Reverse compatibility is particularly useful for

developers of new applications as they can target the library OS with the latest features rather than anticipating adoption curves.

Dynamic Application Relocation

With process serialization and deserialization, a running application needs no longer be tied to the OS instance on which it started. An application could be moved to a new OS instance by serializing its state, moving the state to a new computer as needed, and then deserializing the state to restore the application. Drawbridge enables new opportunities for distributed computing, cycle harvesting, and applications running across OS-servicing reboots.

We are currently experimenting with a distributed computing environment based on Drawbridge in which running applications move from computer to computer to follow the user. For example, an application might start running a user's desktop, then move to the user's smartphone during their commute (which might be facilitated by dynamic instruction-set recompilation), and then move to the user's home computer. Running applications might also move from a laptop or smartphone to the cloud or a local server to conserve battery. While such migration is possible in theory with VMMs, VM images are too large to make it practical.

Process checkpointing has long been used to harvest spare CPU cycles in systems such as Condor [22] by moving background computations to under-utilized workstations. While systems like Condor were limited to applications without interactive user interfaces, with Drawbridge mainstream applications could be moved under the control of a resource manager system to create a "virtual desktop cloud" within an institution. Users could connect to their mobile applications through RDP proxies. Running applications could be cloned to multiple machines or staged to disk, to improve availability and to accommodate hardware failures or periods of peak load. With Drawbridge's per-application granularity, a virtual desktop infrastructure could be created much more cheaply than with VMM-based alternatives.

Serialization and deserialization of running processes offer end users new functionality even if applications are never migrated to a new machine. For example, instead of closing applications for an OS reboot, the applications could be serialized to persistent storage and then restored after the reboot.

Reducing Security Risks

The modern Internet is a place of nearly constant danger. Technical and social attacks often exploit the fact that OS security is oriented toward protecting users from each other, and not from their own software. While the risks of running untrusted software can be mitigated through the use of "sandbox VMs", in practice the resource and administrative costs of a VM image are generally too high to afford each application its own sandbox, so typical users do not employ VMs to isolate software [19]. With the substantially lower overheads of the Drawbridge library OS design, every desktop application could be run within its own sandbox. Running a sandbox per application would mitigate many exploits that attack specific application weaknesses because the exploit could damage at most the single application in the sandbox. Drawbridge might also reduce a large class of social exploits—those which trick users into downloading programs—if downloaded programs always ran within a sandbox with a read-only copy of the library OS.

Cloud hosting services, such as EC2 and Windows Azure, might use the library OS design to substantially lower their per-sandbox costs. While VMMs offer the benefits of a complete OS, and thus will likely always have their place in server consolidation, the library OS uses far fewer resources and thus offers lower costs, particularly for cloud applications with low CPU utilization.

9. Conclusion

This paper provides the first existence proof that a feature-rich operating system can be refactored to produce a library OS that is compatible with desktop applications. Drawbridge provides a Windows 7 library OS that can run a large set of rich desktop and server applications such as Excel 2010, PowerPoint 2010, Internet Explorer 8, and IIS 7.5. With the exception of changes to licensing code, Windows 7 applications run in Drawbridge using unmodified binaries.

We described our heuristics for separating the "OS personality", as captured in the API implementation layer, from the rest of Windows 7 to produce a library OS that is $1/50^{th}$ the size of the full desktop OS, yet still provides rich application compatibility. We believe our techniques are efficient, as we produced the entire Drawbridge system in less than two person-years, and that the same techniques could be applied to other operating systems.

We described our implementation and demonstrated through experiments that Drawbridge can provide increased system security, more aggressive system evolution, and greater application mobility through process serialization and deserialization. We have provided experimental data showing that the library OS design offers significant scalability advantages over VMMs. We demonstrated the improved flexibility for independent evolution by running the same library OS across four host OS releases.

The library OS is very relevant today. It offers new opportunities to significantly improve end-user computing: redefining application packaging, increasing application compatibility across OS releases, enabling running applications to persist across OS reboots and relocations, and reducing the security risks of running untrusted software.

Acknowledgements

We wish to thank our colleagues at MSR, including Andrew Baumann, Barry Bond, David Molnar, and Ed Nightingale who have improved this work by contributing code, experience, and ideas. The anonymous reviewers and our shepherd, Orran Krieger, provided valuable feedback that significantly improved the presentation of our work.

References

[1] Amazon. *Amazon Elastic Compute Cloud (EC2)*. Seattle, WA, 2006.

[2] Ammons, G., Appavoo, J., Butrico, M., Da Silva, D., Grove, D., Kawachiya, K., Krieger, O., Rosenburg, B., Van Hensbergen, E. and Wisniewski, R.W. Libra: A Library OS for a JVM in a Virtualized Execution Environment. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, 2007.

[3] Anderson, T.E. The Case for Application-Specific Operating Systems. In *Proceedings of the 3rd Workshop on Workstation Operating Systems*, 1992.

[4] Appavoo, J., Auslander, M., Da Silva, D., Edelsohn, D., Krieger, O., Ostrowski, M., Rosenburg, B., Wisniewski, R.W. and Xenidis, J. Providing a Linux API on the Scalable K42 Kernel. In *Proceedings of the 2003 USENIX Annual Technical Conference*, 2003.

[5] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[6] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A. and Singhania, A. The Multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[7] Bhattiprolu, S., Biederman, E.W., Hallyn, S. and Lezcano, D. Virtual servers and checkpoint/restart in mainstream Linux. *SIGOPS Operating Systems Review*, *42* (5), 2008.

[8] Bugnion, E., Devine, S., Govil, K. and Rosenblum, M. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Transactions on Computer Systems*, *15* (4), 1997.

[9] Chen, H., Wagner, D. and Dean, D. Setuid Demystified. In *Proceedings of the 11th USENIX Security Symposium*, USENIX Association, 2002.

[10] Cheriton, D.R. and Duda, K.J. A Caching Model of Operating System Kernel Functionality. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, 1994.

[11] Douceur, J.R., Elson, J., Howell, J. and Lorch, J.R. Leveraging Legacy Code to Deploy Desktop Applications on the Web. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[12] Eiraku, H., Shinjo, Y., Pu, C., Koh, Y. and Kato, K. Fast Networking with Socket-Outsourcing in Hosted Virtual Machine Environments. In *Proceedings of the 24th ACM Symposium on Applied Computing*, 2009.

[13] Engler, D.R., Kaashoek, M.F. and O'Toole, J., Jr. Exokernel: an Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.

[14] Franke, H., Russel, R. and Kirkwood, M. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *Proceedings of the Ottawa Linux Symposium*, 2002.

[15] Garfinkel, T. Traps and Pitfalls: Practical Problems in System Call Interposition based Security Tools. In *Proceedings of the Network and Distributed Systems Security Symposium*, 2003.

[16] Gerard Malan, R.R., David Golub, and Robert Brown. DOS as a Mach 3.0 Application. In *Proceedings of the USENIX Mach Symposium*, 1991.

[17] Gupta, D., Lee, S., Vrable, M., Savage, S., Snoeren, A.C., Varghese, G., Voelker, G.M. and Vahdat, A. Difference Engine: Harnessing Memory Redundancy in Virtual Machines. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[18] Helander, J., *Unix under Mach: The Lites Server*. Helsinki University of Technology, Helsinki, 1994.

[19] Howell, J., Hunt, G.C., Molnar, D. and Porter, D.E., *Living Dangerously: A Survey of Software Download Practices*. MSR-TR-2010-51, Microsoft Research, 2010.

[20] Keetch, T., *Escaping from Protected Mode Internet Explorer – Evaluating a potential security boundary*. Verizon Business, London, UK, 2010.

[21] Leslie, I., McAuley, D., Black, R., Roscoe, T., Barham, P., Evers, D., Fairbairns, R. and Hyden, E. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas In Communications*, *14* (7), 1996.

[22] Litzkow, M., Tannenbaum, T., Basney, J. and Livny, M., *Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System*. University of Wisconsin-Madison, 1997.

[23] Loscocco, P. and Smalley, S. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the 2001 USENIX Annual Technical Conference*, 2001.

[24] Love, R. Get on the D-BUS. *Linux Journal*, 2005.

[25] Microsoft. *Internet Information Services 7.5*. Redmond, WA, 2009.

[26] Microsoft. *Microsoft Application Virtualization (App-V)*. Redmond, WA, 2006.

[27] Microsoft *Performance Tuning Guidelines for Windows Server 2008 R2*, Redmond, WA, 2009.

[28] Microsoft, *Remote Desktop Protocol: Basic Connectivity and Graphics Remoting Specification*. Redmond, WA, 2010.

[29] Price, D. and Tucker, A. Solaris zones: operating system support for server consolidation. In *Proceedings of the Large Installation Systems Administration Conference*, 2004.

[30] Roscoe, T., Elphinstone, K. and Heiser, G. Hype and virtue. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, 2007.

[31] Sapuntzakis, C., Brumley, D., Chandra, R., Zeldovich, N., Chow, J., Lam, M.S. and Rosenblum, M. Virtual Appliances for Deploying and Maintaining Software. In *Proceedings of the Large Installation Systems Administration Conference*, 2003.

[32] Soltesz, S., Pötzl, H., Fiuczynski, M.E., Bavier, A. and Peterson, L. Container-based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ACM, 2007.

[33] Spear, M.F., Roeder, T., Hodson, O., Hunt, G.C. and Levi, S., Solving the Starting Problem: Device Drivers as Self-Describing Artifacts. In *Proceedings of the EuroSys 2006 Conference*, Leuven, Belgium, 2006.

[34] Stokely, M. and Lee, C. *The FreeBSD Handbook 3rd Edition, Vol. 1: User's Guide*. FreeBSD Mall, Inc., Brentwood, CA, 2003.

[35] Sugerman, J., Venkitachalam, G. and Lim, B.-H. Virtualizing I/O Devices on VMware Workstations Hosted Virtual Machine Monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference*, 2001.

[36] Torre, C. Mark Russinovich: Inside Windows 7. *Channel 9*, Redmond, WA, January, 2009.

[37] VMWare. *ThinApp*. Palo Alto, CA, 2008.

[38] Waldspurger, C.A. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, 2002.

[39] Whitaker, A., Shaw, M. and Gribble, S.D. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, 2002.

[40] Yee, B., Sehr, D., Dardyk, G., Chen, J.B., Muth, R., Orm, T., Okasaka, S., Narula, N., Fullagar, N. and Inc, G. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, 2009.

[41] Zeldovich, N., Boyd-Wickizer, S., Kohler, E. and Mazières, D. Making information flow explicit in HiStar. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2006.