Composing OS Extensions Safely and Efficiently with Bascule

Andrew Baumann* Dongyoon Lee† Pedro Fonseca‡ Lisa Glendenning§
Jacob R. Lorch* Barry Bond* Reuben Olinsky* Galen C. Hunt*

*Microsoft Research †University of Michigan ‡Max Planck Institute for Software Systems §University of Washington

Abstract

Library OS (LibOS) architectures implement the OS personality as a user-mode library, giving each application the flexibility to choose its LibOS. This approach is appealing for many reasons, not least the ability to extend or customise the LibOS. Recent work with Drawbridge [29] showed that an existing commodity OS (Windows 7) could be refactored to produce a LibOS while retaining application compatibility.

This paper presents Bascule, an architecture for LibOS extensions based on Drawbridge. Rather than relying on the application developer to customise a LibOS, Bascule allows OS-independent extensions to be attached at runtime. Extensions interpose on a narrow binary interface of primitive OS abstractions, such as files and virtual memory. Thus, they are independent of both guest and host OS, and composable at runtime. Since an extension runs in the same process as an application and its LibOS, it is safe and efficient.

Bascule demonstrates extension reuse across diverse guest LibOSes (Windows and Linux) and host OSes (Windows and Barrelfish). Current extensions include file system translation, checkpointing, and architecture adaptation.

1. Introduction

A *library OS* (LibOS) is a user-mode library that runs in the same address space as an application and implements the OS personality on which the application depends. A LibOS architecture gives applications many useful abilities, e.g., to use custom OS implementations [7], to exercise fine-grain control over OS resource management [14], and to achieve strong resource isolation [24]. Drawbridge is a recent LibOS system providing secure isolation while maintaining strong compatibility for existing Windows applications with low overhead [29]. Our aim is to add *extensibility* to these properties.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Eurosys'13 April 15-17, 2013, Prague, Czech Republic
Copyright © 2013 ACM 978-1-4503-1994-2/13/04…$15.00

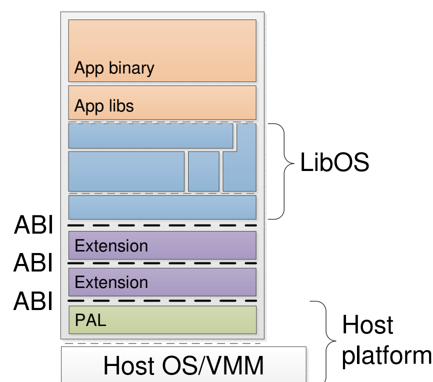

Figure 1. Bascule architecture, showing two extensions

Prior work showed how an application developer could customise a LibOS for substantial increases in performance [20]. However, modifying the LibOS precludes applying the OS vendor's updates and patches, and thus compromises security and maintainability. We focus instead on allowing an end user or system integrator to alter the runtime behaviour of an application with one or more extensions without modifying the LibOS. For example, a user may give an application greater fault tolerance by using checkpoints and migration, even though the OS does not include these features.

Our goal is to enable extensions that are *safe* to admit even if they may be buggy or insecure; *composable* so that multiple extensions may be used together; *independent* of the application, LibOS, and host platform; and *efficient*. The key challenge in achieving this is a tension between efficiency, which leads us to run extensions in-process, and composability of extensions sharing an address space.

To satisfy this goal, we built Bascule, a LibOS extension architecture derived from Drawbridge. It exploits *lightweight interposition* [16, 19] on a small, complete, and stable binary interface, as illustrated in Figure 1. This interface, the *Bascule ABI*, consists of primitive OS abstractions (e.g., threads, files, and virtual memory) and is explicitly designed to be nestable, allowing us to achieve composability and OS independence. As a result, a single extension binary can support multiple host platforms, including Windows and

Barrelfish, and multiple guest LibOS personalities, currently Windows and Linux, with late-binding of extensions at application startup time.

Bascule meets our safety and efficiency goals by running extensions in the same process as the application and LibOS. This protects the system and other applications from buggy or malicious extensions, and allows low interposition overhead by avoiding address space crossings. However, it requires a careful ABI design to permit arbitrary nesting of thread-local storage and exception handlers in the shared address space, as well as a protocol for avoiding address conflicts between the application, LibOS, and extensions.

The contributions of this work are: (*i*) the Bascule ABI, which permits arbitrary nesting of LibOS extensions in the same process; (*ii*) the design and implementation of the LibOSes and host platforms demonstrating the generality of our approach; and (*iii*) an evaluation with extensions including checkpointing, tracing and file system remapping showing that Bascule meets our goal of supporting safe, efficient, composable and independent extensions.

The rest of the paper is structured as follows. §2 further motivates the need for OS extensions with some use case examples, and describes the problems in achieving them with today's software stack. §3 presents Bascule's architecture. §4 describes our implementation of this architecture and of several extensions leveraging it. These extensions enable checkpointing, architecture compatibility, and debugging. §5 evaluates how Bascule meets our goals. Finally, §6–§8 discuss future work and related work, and conclude.

2. Motivation and background

In this section, we first outline some motivating use cases from the literature, some of which we have implemented as Bascule extensions. We focus on general techniques that offer additional functionality rather than just performance increases. We then describe opportunities for implementing such extensions in the commodity system software stack, and identify their drawbacks.

2.1 Use cases for OS extensibility

Checkpointing The ability to checkpoint and restore the complete execution context of an application and its operating environment enables many scenarios, including full persistence in the case of hardware failure [10], migration between host machines [8], and application undo/roll-back.

Despite much research on application-level checkpointing and migration [26], today this functionality is almost exclusively the domain of VMs. A significant reason for this shift is the complexity and granularity of application-level migration: Applications share a rich and complex interface with the OS that tends to maintain implicit state both within the OS and among applications. Migration at the VM level more easily captures this state in a single mechanism.

As we describe in §4.2, Bascule supports checkpointing by an extension since it similarly captures all application and OS state within the user process.

Platform adaptation A widely-used feature is adapting system interfaces for backwards compatibility. For example, the 64-bit extension of the x86 architecture supports 32-bit execution, and most 64-bit PC operating systems continue to support 32-bit program binaries. Besides the architectural support required to switch execution modes, this requires the OS to translate system calls and their arguments between 32- and 64-bit ABIs, at a cost of some added complexity in the OS. Linux supports 32-bit system calls using in-kernel translation code, while Windows relies on a user-mode compatibility layer [25] containing thunks that convert parameters and that switch from 32-bit to 64-bit execution modes.

We implemented several Bascule extensions for architecture adaptation, described in §4.2. These exploit the narrow interface between LibOS and host to reduce the complexity of such adaptation compared to alternative approaches.

Speculation Speculative execution involves predicting the results of a long-running operation then continuing to execute while the operation completes. If the results of the operation were mispredicted, execution is rolled back to the state just prior to speculation. OS-level speculation, where an application speculates across long-running system calls, has been shown to be highly effective at hiding latency for synchronous I/O operations, such as network file systems [27] and local disk writes [28].

An implementation must be able to checkpoint application context at speculation time, and to later roll back execution to that point without visible side effects. This requires complete control over an application's interaction with the outside world. This makes the implementation complex and typically requires deep integration with the OS; for example, the original Speculator system required 7,500 lines of code changes to Linux 2.4 [27].

Bascule could support an extension for speculative execution, because it provides two key properties. First, Bascule permits an extension complete control over the application's interaction with the outside world, allowing it to delay issuing I/O calls while speculating. Second, as demonstrated by the checkpointer, extensions have full visibility and control of the application's state. We note that applying speculation at the level of a single application as in Bascule departs from prior OS-level implementations; we would expect a reduction in complexity as a result of both this and the simpler Bascule ABI.

Record and replay Application record and replay systems have a wide variety of uses including debugging [22, 32], fault tolerance [6], and determinism [9]. These systems involve logging non-deterministic events when recording, then reproducing the same events at replay time. This requires control over all sources of non-determinism visible to the

application under record or replay, and has been implemented at both the OS [23] and VM [13] level. In both cases, much like speculation, the implementation involves significant changes to an OS or VM monitor to log events and checkpoint state. Bascule would support such functionality through interposition on the ABI, which permits capturing of non-deterministic inputs and checkpoint/rollback.

2.2 OS extension mechanisms in today's stack

The primary mechanism for extensibility in today's commodity operating systems is loadable kernel modules. Some popular extensions implemented as modules include fast web servers and network file servers. However, implementing extensions in the kernel has several problems, not least safety, since the code for a kernel module must be fully trusted by all users and applications, and portability, since modules are tightly coupled to the internal interfaces (and in some cases even the specific version) of an OS. We therefore seek a more flexible solution.

Some of the extensions enabled by Bascule, including checkpointing and architecture adaptation, are provided by virtual machines. However, as observed by others [29, 30], virtual hardware makes a poor interface to what is essentially another OS. This complicates the implementation of many extensions, since low-level hardware interfaces such as virtual disks and page tables lack much of the semantic information present at the abstract level of files and memory mappings. Moreover, compared to a virtual machine interface, Bascule extensions need not concern themselves with privileged instructions, additional processor context and data structures (such as page tables), and virtual devices. This leads to reduced complexity and resulting efficiency gains. For example, Bascule checkpoints are substantially smaller than a typical virtual machine checkpoint.

One attractive extension technique is interposition, which involves capturing events crossing an interface, then modifying, dropping, or injecting those events before passing them on to the underlying implementation. Past work has argued the benefits of interposition as an extension mechanism [16, 19, 33]. These benefits can include: (i) decoupling of extensions from the host, since extension authors must only handle the interface between applications and the host rather than internal kernel interfaces; (ii) transparency, to both applications and the host; and (iii) composability, since extensions can be invoked in a chain for each event.

However, today's commodity software stack (Figure 2) lacks a stable "thin waist" interface at which extensions can be implemented through interposition without depending heavily on the internal details of either an OS or VMM. In each case of OS API, system call ABI, and virtual hardware, the interface is under-specified, complex, and evolving. This motivates our use of a new interface designed for interposition between LibOS and host, as we describe in the following section.

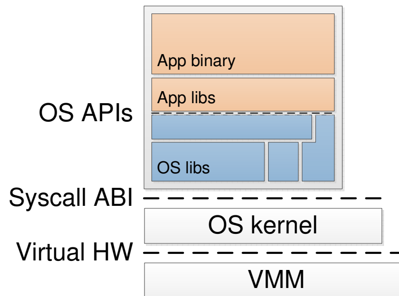

Figure 2. Current system software stack

3. Bascule architecture

Two lessons from related work motivate our design. First, like virtual machines, we admit a tight coupling between the OS and application for the purposes of compatibility and do not attempt to change this interface. However, by basing Bascule on Drawbridge [29], we enable the OS to run as a library in the application's address space. Second, past work with Exokernels [20] showed that a user-mode library OS permits application-specific OS extensibility, since the LibOS is under the control of the application, and each application can use its own LibOS. We exploit this property, but do not require that the LibOS be modified or customised for extensibility, since an unmodified LibOS can receive security patches directly from the OS vendor. Instead, we enable late-binding of extensions through use of interposition on a stable and complete binary interface consisting of primitive OS abstractions such as files and virtual memory.

This interface, the *Bascule ABI* shown as a dashed line in Figure 1, is accessed through a table of function entry points. It is ultimately provided by the platform abstraction layer (PAL), a library of code injected by the host platform at startup time. We typically refer to the host platform as an OS, however it may also be a bare virtual machine [29]; the PAL ultimately invokes the host through system calls or hypercalls, however it may also implement significant functionality in-process. Between the PAL and LibOS are one or more extensions chosen at load time. Since each extension both provides and consumes the Bascule ABI but has no other dependencies on its execution environment, extensions can be stacked arbitrarily, composing functionality. Within this stack of LibOS, extensions and PAL, each extension layer interacts with its *guest*, which may be either another extension or the LibOS, and its *host*, again either another extension, or the PAL.

Since extensions run in the application address space, and interpose on the ABI through a table of code pointers, they are both lightweight and safe to the limits of the host's confinement mechanism: an extension can do no more than the application (or LibOS) itself. This restricts the functionality that can be implemented; an extension cannot reliably confine an application (since its execution could be compromised by the application) nor directly coordinate the activ-

ity of multiple applications. Nevertheless, interposition on primitive abstractions between a LibOS and host platform enables a wide range of functionality, including (we believe) the motivating examples from §2.1.

In contrast to a typical OS process, Bascule loads significantly more code (in the form of the LibOS, extensions and PAL), much of it developed independently, into the process address space. This is potentially problematic, because an address conflict between any of these components would either prevent an application from running, or (worse) lead to hard-to-diagnose memory corruption and crashes. To address this, we define a protocol for managing use of virtual address space by extensions and the PAL, and restrict fixed address allocations only to the application and its LibOS – all extension code must be position-independent, and all memory allocations made by extensions must reside in a region allocated to the extension at process startup. Our experience is that these simple techniques, combined with a large (64-bit architecture) virtual address space and widespread adoption of address-space layout randomisation techniques [5] (which require position-independent code to be effective), have avoided any problems with virtual address conflicts.

In the remainder of this section, we describe the Bascule ABI in more detail, as well as the protocols for process startup and initialisation of extensions.

3.1 ABI

Figure 3 summarises the Bascule ABI, which is derived from the Drawbridge [29] ABI with key modifications we describe below. The Bascule ABI, like the Drawbridge ABI, is based on a small set of primitive OS abstractions such as threads, files, I/O streams, and virtual memory mappings. These abstractions are chosen to have well-defined semantics that are easily supported by a range of OS implementations, and permit the host to expose virtualised resources to the LibOS with minimal duplication of effort. The simple abstractional nature of this ABI makes it an effective platform for implementing extensions. Furthermore, because the set of abstractions was deliberately chosen to mimic those in commodity OSes, we were able to construct a Linux LibOS (described in §4.1) without duplicating functionality.

While our aims in designing the Bascule ABI were generality and nesting, we admitted some pragmatism for reasons of performance and implementation expedience. Of particular concern was finding compromises between generality and efficient implementations on existing host OSes such as Windows in problematic areas including exception handling and thread-local storage, as we discuss below. Further, while our goal and expectation is for the Bascule ABI to remain stable, it carries an explicit version number, permitting us to add or extend functionality in the future if required, and even support extensions or guests built for old versions of the ABI through "version adapter" extensions that rewrite calls to conform to the new ABI.

To enable its goals of OS independence and lightweight, composable extensions, and in particular to support arbitrary nesting of extensions, Bascule changes the following aspects of the Drawbridge ABI:

Calling conventions and C ABI We codified the (previously implicit) calling convention for ABI calls, by specifying the register allocation and stack frame layout for arguments to mimic the ABI used by the Microsoft C compiler. This specifies the exact registers and stack frame layout to be used for passing and returning parameters from ABI calls, and layout of shared in-memory data structures. It enables the use of different compilers and runtimes for each layer in the Bascule stack (application/LibOS, extensions and PAL) with the use of small thunks for making or handling calls across an ABI boundary.

Stack use across ABI calls Since compiler and OS behaviour differs widely in regards to a thread's stack, we restrict a callee across the Bascule ABI boundary from making any assumptions about the location, size, bounds or layout of the caller's stack. Instead, a caller's stack is immutable to the callee, who is responsible for allocating and switching to a private stack if required. An opaque per-thread parameter, which is established at thread creation time and must be explicitly passed on each ABI call may be used for this purpose – typically the incoming call thunk will load a stack from this parameter before calling into C code. The stack pointer is also undefined for all three upcalls in the Bascule ABI: process startup, thread startup, and exception delivery. We designed these calls so that all arguments are passed in registers only, permitting the callee to load their own stack while avoiding the need for the caller to modify it.

Exception handling OS APIs differ on their behaviour in response to exceptions, such as invalid instruction, memory access fault, software breakpoint, divide by zero, etc. Drawbridge relies on a mechanism derived from Windows: upon detecting an exception, the host OS writes a data structure describing its cause and the full processor context at the time of the exception onto the thread's stack, then causes the thread to jump to a user-mode upcall function, passing the exception record as a parameter. The upcall function is then free to attempt to ignore the exception by restoring the context and resuming execution, or take other corrective action, such as running an exception handler. This model works well for Windows, but is problematic for a general OS-independent ABI like Bascule, since it requires knowledge of stack layout and bounds, and is likely to break other platform ABIs that permit use of memory beyond the current stack pointer. Furthermore, Bascule extensions (such as the checkpointer we describe in §4.2) may wish to interpose on exceptions before they are delivered to the application.

In Bascule, the exception record is saved (in a portable format defined by the ABI) in a region of memory under the control of the PAL, and an upcall is delivered to the next

Downcalls:

```
VirtualMemoryAllocate(Addr, Size, Type, Prot) -> Addr
VirtualMemoryFree(Addr, Size, FreeType)
VirtualMemoryProtect(Addr, Size, Prot)
SemaphoreCreate(InitCount, MaxCount) -> SemaphoreHandle
SemaphoreRelease(SemaphoreHandle, ReleaseCount)
SemaphorePeek(SemaphoreHandle) -> Count
NotificationEventCreate(InitialState) -> EventHandle
SynchronizationEventCreate(InitialState) -> EventHandle
EventSet(EventHandle)
EventClear(EventHandle)
EventPeek(EventHandle) -> State
ObjectReference(Handle)
ObjectClose(Handle)
ObjectsWaitAny(Count, Handles, Timeout) -> Index
ThreadCreate(Routine, Arg, Stack, Params) -> ThreadHandle
ThreadExit()
ThreadYieldExecution()
ThreadRaiseException(ThreadHandle, OpaquePointer)
ProcessCreate(Parameters) -> ProcessHandle, ThreadHandle
ProcessExit(ExitCode)
ProcessGetExitCode(ProcessHandle) -> ExitCode
SystemTimeQuery() -> Time
StreamOpen(URI, DesiredAccess, ShareAccess,
           OpenDisposition, Options) -> StreamHandle
StreamRead(StreamHandle, Off, Size, Buffer) -> AsyncHandle
StreamWrite(StreamHandle, Off, Size, Buffer) -> AsyncHandle
StreamSetLength(StreamHandle, Length)
StreamFlush(StreamHandle)
StreamDelete(StreamHandle)
StreamGetEvent(StreamHandle, EventId) -> EventHandle
StreamRename(StreamHandle, URI)
StreamAttributesQuery(URI) -> Attribs
StreamAttributesQueryByHandle(StreamHandle) -> Attribs
StreamMap(StreamHandle, Addr, Flags, Prot, Offset, Size)
StreamMapPeBinary(StreamHandle, Addr, Flags)
StreamUnmap(Addr)
StreamEnumerateChildren(StreamHandle, EnumHandle) -> Name
AsyncPoll(AsyncHandle) -> Results
AsyncCancel(AsyncHandle)
RandomBitsRead(Buffer, Size)
InstructionCacheFlush(Addr, Size)
```

Upcalls:

```
InitializeProcess(StartParams, InitData)
ThreadRoutine(StartParams, Arg)
ExceptionDispatch(ExceptionInfo, HWContext)
```

Figure 3. Functional summary of the Bascule ABI.

layer in the stack (extension or LibOS). Any subsequent exception may re-use this memory, so if the handler wishes to support nested exception handling, it must first copy the exception record to a stable location (for example, its own stack) without raising a subsequent exception. This requires careful defensive programming, akin to critical paths inside an OS kernel that also cannot survive an exception. Each layered extension in the stack is responsible either for handling its own exceptions or passing them on to the layer above, thereby enabling full interposition on exception upcalls.

Thread-local storage Most OS platforms provide an efficient mechanism for accessing thread-local data, and/or rely on one for their own libraries, and for compatibility Bascule must support this. Typically, this takes the form of a reserved register which refers to a per-thread data structure.

As described, this is straightforward for Bascule to support: the thread registers would simply need to be preserved across ABI calls, permitting each layer in the stack to choose its own mechanisms for thread-local storage. However, the x86 architecture adds a serious complication: for legacy reasons, all OSes with which we are familiar (including Windows and Linux) access thread-local state through a segment register such as FS or GS. Unlike a general-purpose register, x86 segment registers can only be loaded in one of two ways: indirectly, by referring to a segment descriptor in a kernel-controlled table (which also restricts the base address to 32-bits) or directly by the kernel, using a privileged `wrmsr` instruction.[1]

The practical upshot of this limitation is that a general nestable ABI for thread-local storage is impossible on x86. In the general case, it would require system calls to reload FS/GS on each ABI crossing, however, even if we were willing to pay the steep performance penalty this implies, those system calls are host-specific and could only be reached through ABI downcalls, leading to a chicken-and-egg problem. Instead, we settled on a compromise that supports all the use-cases we have encountered to date: at thread creation time, the guest may request FS and/or GS segments meeting certain constraints (size, alignment, etc.). If the call succeeds and the thread is created, its FS/GS segments will refer to memory that is allocated by the host and exists for the lifetime of the thread. If any intermediate layer (i.e. extension or PAL) uses them, it must either save and restore their contents by copying, or else request a larger segment and use only the space beyond that allocated by the guest LibOS. Also, no thread may access another thread's FS or GS segments, since their contents are not guaranteed to be stable across ABI calls. Overall, this solution is functional for our needs, but inefficient and potentially error-prone; however, given the constraints of current architecture and compatibility, it appears to be the best compromise.

Other changes Finally, Bascule makes a number of functional changes to the calls in the Drawbridge ABI. First,

[1] Intel and AMD have recently (on processors first available in 2012) added unprivileged instructions for manipulating FS and GS registers, however these are not yet widely available and require kernel support to enable.

the ABI was simplified and generalised in a number of ways compared to the previously-published version [29]. For example, all I/O is now asynchronous and only one call (`ObjectsWaitAny`) blocks the calling thread while awaiting an event notification. This simplifies the implementation of extensions that control thread execution, including the checkpointer which must interrupt blocking calls to achieve quiescence. Second, explicit ABI calls for checkpointing are removed; in Bascule, this is provided by an extension (described in §4.2), and does not require specific ABI support. Third, a number of implicit Windows dependencies that were discovered while testing the Barrelfish host and Linux guest were resolved; for example, Windows assumes that virtual memory mappings are aligned to 64kB granularity, and Drawbridge silently enforced this requirement in its `VirtualMemoryAllocate` and `StreamMap` ABIs. Finally, we found that the socket interface between network stack and applications is too rich and variable between OSes to be supported by a simple abstraction, and thus are in the process of switching to a lower-level virtual network interface and moving the network stack inside the LibOS for greater compatibility with existing applications.

3.2 Packaging and startup

Like Drawbridge, Bascule makes use of declarative manifests that specify the application and library OS components to be loaded, their dependencies, and the host services to which they have access. Bascule also uses manifests to configure extensions. An extension's manifest identifies the binary module to be loaded for that extension, along with any dependencies for the extension, and configuration meta-data, such as virtual address space requirements.

Extensions are packaged as relocatable libraries in PE format, however they have with no external dependencies, nor import or export any symbols. This does not preclude the use of further dynamic libraries nor the use of an alternative executable format, however, since an extension is free to load further images to which it has access at process startup time.

At startup time, the PAL is loaded in a platform-specific manner. Thereafter, each layer (PAL, extension or LibOS) in the stack is responsible for loading and initialising the layer immediately above it using only the ABI; so, the PAL loads the last extension and passes control to it, which in turn loads the next extension, and so on until the library OS is loaded, which ultimately loads and begins executing the application. This ensures that of the entire stack, only the PAL depends on the host OS, while extensions are free to use the ABI for their own initialisation as they require, before loading further layers. In order to load a layer, its module file is first loaded into memory and relocated, then its an upcall function (identified using the entry point field in the PE header) is executed. This upcall takes as a parameter a table of downcall function pointers implementing the ABI, provided by the layer below, and is required to fill in a table of upcall pointers. This permits each layer to selectively interpose on various ABI calls, and make use of the ABI implementation provided by the layer beneath it.

A binary blob is also passed to each extension's startup function; this contains parameters specific to that extension, in a format defined for it. For example, the parameters to the checkpoint extension are a trigger event to initiate checkpoints and a storage path in which to save them, or a checkpoint file to resume. Generating this data structure requires a small amount of extension-specific logic; we could remove this requirement with a generic mechanism for parsing and passing parameters, however we expect that most extensions will require a user interface for configuration, and the generic specification of these is out of scope for this paper.

To manage use of the shared virtual address space, and avoid possible conflicts between the application, LibOS and extensions, Bascule pre-reserves address space ranges for each extension, which are passed to the extension's startup function. Extensions must be fully relocatable, and only allocate memory within their reserved regions. These are always high in the (64-bit) address space, to avoid any conflicts with fixed allocations that may be made by the application. While this strategy is no guarantee of success, we have not yet encountered an address conflict, and expect to add further metadata to the application and LibOS manifests, identifying known fixed address regions that may be used, and which will then be avoided for loading extensions.

4. Implementation

This section describes the implementation of Bascule, starting with the guest library OS environments, extensions we have implemented to date, and finally describes the current host platforms. Since one of our goals is OS version independence, we have developed several host platforms and guest LibOS personalities in addition to the core Windows platform supported by Drawbridge. These show that using an abstractional interface between the LibOS and host platform does not necessarily limit the scope of OS personalities or hosts for Bascule.

4.1 Implementing library OSes for Bascule

Windows We have reused the Windows LibOSes (for 32- and 64-bit x86) created by the Drawbridge project, adapted for the Bascule ABI. We also fixed a number of accidental ABI violations that were discovered in the process of testing the Barrelfish host described later in §4.3. These were primarily cases where state was leaking into the picoprocess from the host kernel: every NT process has a data structure (`USER_SHARED_DATA`) mapped at a fixed address, and the Drawbridge LibOS was assuming the presence of this structure provided by the host; similarly, we updated the LibOS to create and manage its own per-process and per-thread data structures, rather than relying on those provided by the host kernel. In other respects, the Windows LibOS is unchanged,

and we refer the reader to the Drawbridge paper [29] for further details on its implementation.

Linux To exercise the ABI with a non-Windows guest and check that the abstract Bascule ABI does not restrict the choice LibOS personality, we implemented a prototype Linux-based LibOS. We set ourselves the initial goal of running Linux applications in a LibOS with no further changes to the ABI. We found that by applying similar techniques as used in the Drawbridge project, including reuse of code where possible and pragmatic reimplementation elsewhere, this was achievable: it took one of us only 12 weeks to construct an initial prototype Linux LibOS; while it is far from complete, it already has enough functionality to run major applications, such as Firefox and Apache.

Our Linux LibOS supports 64-bit x86 binaries, and is based on a modified `libc` (eglibc version 2.15) coupled with a system call emulation layer. We lightly modified `libc` and its related libraries including `ld-linux` and `libpthreads`, to replace system call instructions with calls to our emulation layer. This limits our current LibOS to applications that dynamically link a compatible version of `libc`, however we expect that these changes could also be performed dynamically through trap reflection and/or binary rewriting. The bulk of the new code resides in the system call emulation layer, which emulates the Linux system call API by using the underlying Bascule ABI. Our emulation layer performs a similar task to existing OS compatibility layers [11, 17, 34], however because it targets a common intermediate ABI rather than a specific host OS API, the same LibOS binaries work across all Bascule host platforms.

At process startup time, we use a simple ELF loader which is packaged as a PE binary (as required by the Bascule ABI) to load the emulation layer and the native Linux loader `ld-linux.so`. This in turn then loads the application, including its libraries and our modified `libc`, and starts its execution. The emulation layer includes thunks for shuffling register arguments as appropriate when issuing Bascule ABI downcalls, but is otherwise compiled as a typical Linux shared library. It presently implements 48 system calls completely, and 33 only partially. While this is a fraction of the total count of more than three hundred system calls, it is enough to run real applications and exercise the ABI.

While we were able to meet our goal of no ABI changes, several unanticipated quirks required workarounds. First, we discovered that the standard Linux/ELF ABI for thread-local storage uses negative (sign extended) offsets from the FS segment. At present the Bascule ABI has no way to express this requirement, so we have made minor modifications to the host platforms to ensure it is met. Second, due to a mismatch between alignment constraints in the Linux API (4kB) and Windows kernel memory manager (64kB), we are presently unable to implement a true `mmap()` on all host platforms, and instead emulate it by allocating virtual memory and performing stream I/O. We expect ultimately to make some changes to the ABI to improve performance in these areas.

We also intend to improve the compatibility of our LibOS, implementing further system calls and porting relevant code from the Linux kernel. Significant unsupported functionality currently includes multiple address spaces and `fork`, signals, and advanced networking APIs beyond basic TCP sockets. Of these features, only `fork` will require ABI changes, as we discuss in §6. For GUI applications, we presently assume the presence of an X server outside the LibOS which is accessed through a socket, but also anticipate moving this in-process with the LibOS.

The emulation layer and loader stub presently account for 13.5k lines of C code, primarily new code in the emulation layer. We also changed 2.8k lines of code in eglibc to redirect system calls to our emulation layer. Our application binaries, supporting libraries, and configuration files are all taken as-is from an Ubuntu 12.04 installation image. To date we have been forced to make only one change to an application, to ignore a failing `fork()` call in Firefox during startup.

4.2 Extensions: using Bascule to extend LibOSes

We have implemented several Bascule extensions to date, including call tracing, file system remapping, checkpointing, and a number of extensions for architecture adaptation and emulation, which we describe here.

Tracing

The tracing extension is a simple demonstration of extension functionality, which also serves as a useful debugging tool. It is implemented in approximately 1000 lines of C code, and logs ABI calls, their parameters, and results, but makes no other changes to the execution behaviour of the layers above it. It can be configured to log all calls, specific functionality (e.g. stream I/O), or only calls whose result code indicates a failure. The log is either written to an I/O stream, or maintained in memory where it may be inspected by a debugger.

As with all other extensions, the tracer is packaged as a self-contained dynamic library (DLL) with no external dependencies. At process startup time, it is loaded and its entry point is invoked by the layer (PAL or extension) beneath it in the stack. The parameters passed to its entry point include the ABI call table for the layer that loaded it, and a parameters structure which includes parameters to the extension and also identifies the subsequent layers to be loaded. The tracer fills out the upcall table from the layer below (providing a pointer to its exception handler), and initialises a downcall table that it will provide to the layer above it. It then loads this layer, using ABI calls such as `StreamOpen` and `StreamMapPeBinary`, and runs its entry point passing the appropriate arguments. Subsequent ABI calls from the layers above will pass through the tracer's call handlers, which issue the call to the layer beneath before logging the results.

To ease construction of extensions such as the tracer, we make use of several libraries that are statically linked into the

final extension DLL. These include some minimal C runtime support, assembly thunks for wrapping ABI call handlers, and a memory allocator for stacks. As required by the ABI, a callee must not modify the caller's stack, so each extension typically interposes on `ThreadCreate` and `ThreadExit` to allocate its own stack for that thread. This adds a small memory overhead for each thread in each extension, but for simple extensions such as tracing the stack space required is extremely small. This could be avoided, with a significant increase in complexity, either by dynamic reuse of stacks between threads or an ABI-level protocol to negotiate use of the caller's stack. The assembly code used when switching stacks contains sufficient debug information for a debugger to display one unified stack backtrace for the thread.

File system remapping

This extension implements a translation layer between file paths used by the application and LibOS and those provided by the host system, and illustrates the use of Bascule to correct bugs or undesired behaviour in an application. For example, the extension can be used as a workaround for hard-coded file paths in the application, to give the application a private (writable) copy of various system files, or to discard writes to certain files (e.g., to quieten needless logging).

The extension works by interposing on the `StreamOpen` and `StreamRename` ABI calls and rewriting file paths according to the mappings specified in its configuration. The restricted nature of the Bascule ABI makes this extension straightforward to implement compared to a traditional OS, which may have many different system call interfaces where file paths are used or exposed, and additionally the same extension binary works across all Bascule host platforms.

Checkpointing

In Drawbridge, application checkpoint and restore functionality were implemented within the LibOS, with some support from the host [29]. However, there is no fundamental reason for this functionality to be part of either the host or the guest, so Bascule instead implements it as an extension independently of both. The checkpointer extension tracks the use of the process' virtual address space, and implements a translation layer between resource handles provided by the host and those visible to the guest. It also has two control channels back to the host: an event is signalled to initiate a checkpoint, and startup parameters indicate whether to start the application from scratch or restore a checkpoint.

At checkpoint time, the checkpointer opens a file (using the ABI) and serialises to it all regions of the address space created by (allocated or mapped) the layers above it. This includes both the guest LibOS and application, and any extensions which may be loaded in the stack above the checkpointer. It also serialises the metadata it is tracking for each open handle, and thread contexts, including the stacks of any threads presently executing inside the checkpointer. It does not serialise other state in the process, such as the PAL or extensions below it in the stack. This permits a guest to be checkpointed and later restored, either on a different host platform or with a different set of extensions loaded beneath the checkpointer.

When starting, if the checkpointer has been requested to resume a previous checkpoint, it opens the checkpoint file, recreates the open handles by replaying relevant ABI calls (such as `StreamOpen`) and updating its private handle translation table, restores the address space mappings and recreates and resumes the threads. In this way, other Bascule layers can be and are completely unaware of checkpoint functionality, and the same checkpoint layer binary works equivalently for Linux and Windows guests, and Windows and Barrelfish hosts.

The checkpointer extension consists of 14.5k lines of C and assembly code. Beyond one-shot checkpoint and restore, we are also investigating continuous incremental checkpointing in support of a replication mechanism in the style of Remus [10], but do not discuss it further here.

In contrast to a VM-based approach, the advantage of performing checkpointing at the Bascule level is simplicity, and resulting efficiency gains. In Bascule, the guest consists solely of user-mode code, and interacts with the host only through the Bascule ABI. Compared to a hardware VM interface, this excludes privileged instructions, additional processor context and data structures (such as page tables), and virtual devices, along with the guest OS code required to support them. As a result, Bascule processes have much smaller memory footprint and checkpoints than a typical hardware VM [29].

Architecture adaptation

Motivated by the vast number of existing x86 applications, we have used Bascule to experiment with running 32-bit x86 Windows applications and an x86 LibOS on alternative host architectures. Two prototype extensions support the 32-bit x86 Bascule ABI atop an alternative architecture host ABI.

32-on-64-bit x86 As discussed in §2.1, the 64-bit extension to the x86 architecture supports backward compatibility with 32-bit code, and most 64-bit OSes support 32-bit applications, either with a modified kernel or custom user libraries. In Bascule, an extension provides this functionality at the ABI level without modifying either the guest LibOS or host OS. This extension translates between the 32-bit and 64-bit variants of the Bascule ABI, switching from the processor's execution mode for each ABI crossing, while converting arguments appropriately for the different word sizes. It also modifies the parameters to ABI calls to ensure that all virtual memory allocations exposed to the guest lie below the 4GB address limit.

x86 on ARM JIT This experimental extension also supports the x86 Bascule ABI, but does so on an ARM host platform. It uses an interpreter and just-in-time (JIT) compiler to convert x86 instructions to ARMv6 code. ABI calls occur-

Figure 4. Experiment demonstrating an x86 Windows application (Reversi) on an ARM phone.

ring within the x86 code are translated and issued as downcalls according to the Bascule ABI. Since it must only support user-mode code, the interpreter is significantly simpler than full-system emulators, as would be required by a VM-based solution. For example, address translation is direct-mapped to the host process address space relying on the host OS to catch invalid memory accesses, and no virtual devices are needed. Upon an exception, the interpreter reconstructs the corresponding x86 state, and delivers the exception to the guest LibOS as an ABI upcall. The entire interpreter, including the JIT engine which adds most of the complexity, is implemented in approximately 50k lines of C code.

Figure 4 shows an x86 application (Reversi) running atop the Windows 7 LibOS on an ARM phone using this extension. The application and LibOS are completely unmodified – the same binaries also run on an x86 host, albeit much faster, using the 64-bit compatibility extension previously described.

4.3 Host platforms: implementing Bascule

Windows The Windows host implementation is based on Drawbridge [29], and supports the x86 (32- and 64-bit) and ARM (32-bit) architectures. It consists of three components: (i) a PAL library that is loaded into the guest picoprocess and implements the ABI, generally by issuing custom system calls for each ABI downcall; (ii) modifications to the host kernel to implement the system calls made by the PAL and confine the guest process by preventing it from invoking other system calls; and (iii) a security monitor that runs in a separate user process, which is consulted by the kernel-mode component for policy decisions (such as stream open calls) and is responsible for starting and managing Bascule applications.

As described in the previous section, we modified Drawbridge to support our ABI and added support to the security monitor to specify extensions to be loaded and pass parameters to them.

Barrelfish Just as we used Linux to test the generality of the Bascule ABI from the perspective of a LibOS personality, we similarly aimed to exercise the ABI with a non-Windows host. Barrelfish [2] is a research operating system project, whose primary goal is to explore OS structure for future multi- and many-core systems. Implementing a Bascule host for Barrelfish has two key benefits. First, it allows us to ensure that Bascule has no hidden dependencies on Windows, since Barrelfish differs significantly from Windows: it builds with GCC and uses the Linux/ELF ABI, its native API is unique but in many respects closer to POSIX than to Win32, and significant portions of functionality that would typically reside in the kernel are implemented in user-mode libraries, including thread scheduling and address space management. Second, it allows Barrelfish to run real (Windows and Linux) applications, adding both functionality and potential workloads for further research.

Unlike the Windows host implementation, on Barrelfish we rely on the system's native confinement mechanisms for security, and implement the ABI entirely in-process: the Barrelfish PAL is also a loader application which links against the native Barrelfish API libraries. At startup, it loads the guest extensions and LibOS into its address space, then jumps to the guest code, implementing all of the Bascule ABI in-process by translating ABI calls to equivalent native Barrelfish functionality. On each downcall and upcall through the ABI, a generated assembly fragment, or "thunk", converts the call from the Bascule ABI calling convention to the native Linux ABI, by shuffling argument registers and switching stacks. The PAL also manages entries in the x86 local segment descriptor table, to provide suitable support for thread-local storage. For convenience, to enable reuse of existing packages, it supports the same packaging mechanism and manifest format as the Windows host's monitor.

The Barrelfish PAL consists of approximately 8k lines of C code. It implements the Bascule ABI with the exception of subprocess support. Nevertheless, it is able to run both Windows and Linux guests, including rich applications such as Excel, PowerPoint, and Apache.

5. Evaluation

The goals for Bascule were *safety* of the host and other applications, *independence* of extensions, LibOSes and host platforms, *efficiency* and *composability* of extensions.

Bascule meets the safety goal by design, so we do not evaluate it here: since extensions run in the guest process address space, they are no more privileged than the guest process itself. The host must ensure that any system calls that are permitted from the guest process are appropriately secured, but since the ABI is narrow, this is a significantly easier task than for a traditional OS API.

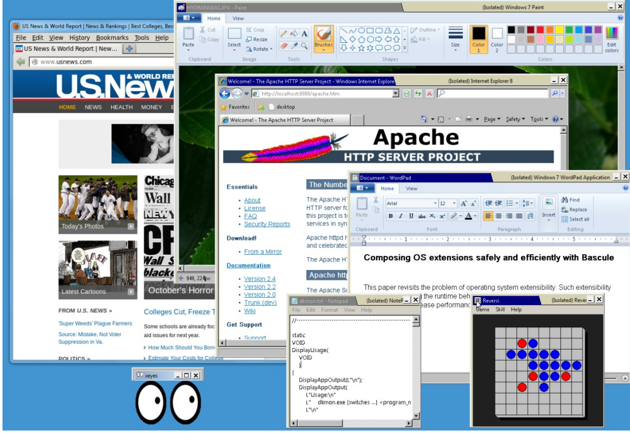

Figure 5. Linux and Windows applications running with their respective LibOSes on the Bascule Windows 7 host.

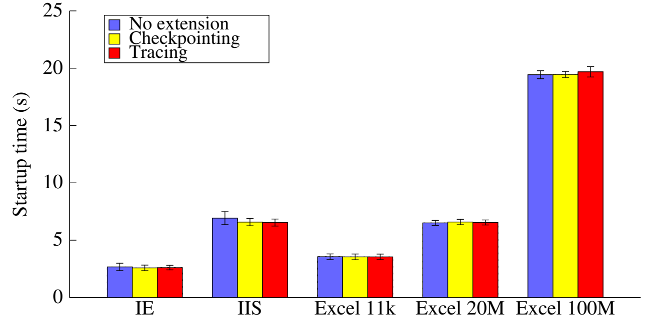

Figure 6. Runtime overhead of extensions.

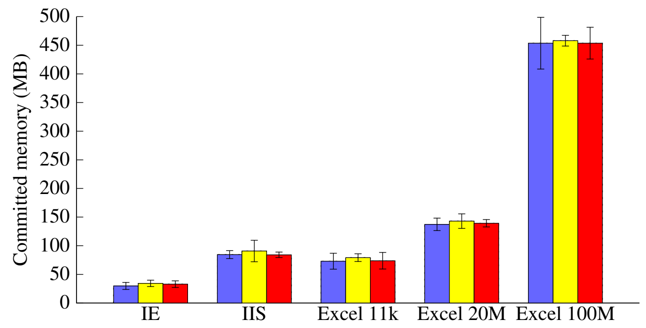

Figure 7. Memory footprint of extensions.

In the this section, we evaluate the remaining goals. We focus on the Windows LibOS and host implementation, as they are the most mature. All reported experiments were run on a Lenovo ThinkStation E30 PC with a 3.2GHz Intel Xeon E31230 4-core processor and 8GB of RAM.

Independence As described in §4, we have implemented host PALs for Windows and Barrelfish, and LibOSes for Windows and Linux. We have run both on both host platforms with a variety of applications – for example, Figure 5 shows a combination of Windows and Linux applications running on the Windows 7 host, including xeyes and Firefox (Linux LibOS), MS Paint, Internet Explorer, WordPad, Reversi and Notepad (Windows LibOS). In this screenshot, IE is displaying a page served by the Apache server, which also runs on the Linux LibOS. We have tested our 64-bit extensions (tracing and checkpointer) with both LibOS guests.

Direct cost of interposition We first used a microbenchmark to measure the overhead of a null extension that interposes on the Bascule ABI but makes no change to any calls. We measured the difference in processor cycles (using `rdtsc`) spent in a Bascule ABI call with and without this extension. We repeated each run 1000 times, and found that the average per-call overhead was 86 cycles (with standard deviation of 29 cycles). This is the cost of switching stacks on call and return, which is minor compared to the overhead of the system call that ABI calls ultimately invoke.

Extensions also impose a memory overhead. This is highly dependent on the specific extension; the base cost is the size of the extension binary image, a few kB of data structures such as the ABI call table and initialisation parameters used at startup time, and an additional per-thread stack that can be as small as a single page.

Runtime overhead of extensions We next measure the overhead of several Bascule extensions on real applications. Figures 6 and 7 show the startup time and committed memory for a range of applications on the Windows LibOS, running with a variety of extensions. For Internet Explorer (IE) and the IIS web server, we measure the time to render a trivial page and serve the first request respectively. For Excel, we measure the time to open one of three sample input files of increasing size. All times and sizes include the application, LibOS, and any extensions. The plots show the mean and standard deviation of 100 runs. We see that our extensions have negligible overhead on real-world applications, both in terms of time and memory footprint.

Overhead of LibOS-independent checkpoints Since the checkpointer extension implements functionality originally developed as part of the Windows LibOS, it allows us to compare the overhead of performing checkpointing in an OS-independent Bascule extension rather than building it into the LibOS. Figures 8 and 9 report the size of checkpoint files and time to checkpoint for the same Excel configurations running on Bascule, using either the checkpoint code built into the Windows LibOS, or our new extension. We plot uncompressed file sizes, but note that checkpoints compress well. For example, a sample checkpoint for Excel with a 20 MB input compresses from over 100 MB down to 22 MB, only slightly larger than the original input file.

Figure 8 shows that Bascule checkpoints are larger and more variable in size than LibOS checkpoints, with the size difference not noticeably affected by the absolute size. The reason for this is as follows. Parts of the address space, including relocated segments of executables and dynamic libraries, are initialised deterministically by the LibOS and

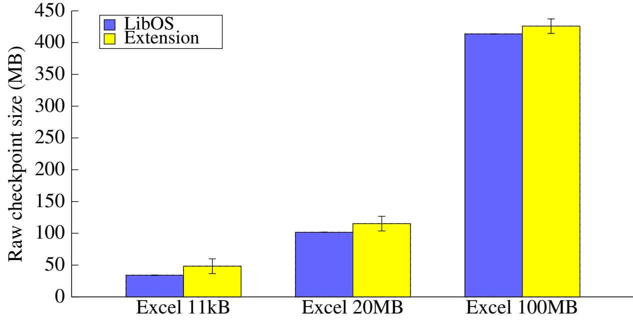

Figure 8. Size of uncompressed checkpoints.

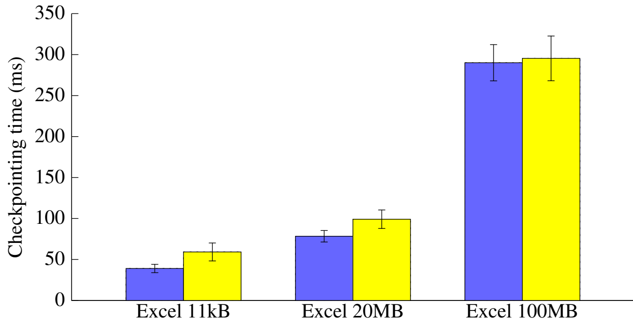

Figure 9. Time to checkpoint.

not included in its checkpoints. For each executable segment, the LibOS applies a deterministic relocation algorithm to rectify internal pointers with the location where the segment was actually loaded. The LibOS checkpointer knows this, and thus it can avoid writing the initialised data to the checkpoint, and just resume from a checkpoint by following the same procedures. By contrast, the Bascule checkpointer must note all page modifications in the checkpoint, including those caused by relocation. For example, the `Excel.exe` text segment is 22 MB. The LibOS only ever checkpoints 24 kB of this, regardless of where it is loaded, whereas the extension must often checkpoint all of it, when it is relocated by the LibOS. This explains the checkpoint size difference. It also explains the size variability, because the extension only needs to checkpoint this segment when address-space layout randomisation (a feature of the host OS) causes it to be loaded elsewhere than its preferred address.

This size variation leads to a somewhat longer time to checkpoint Excel, as seen in Figure 9, because a significant fraction of checkpointing time is spent writing the checkpoint file.

Cost of composability Like independence, composability is partly a functional property achieved by careful design of the Bascule ABI to enable nesting. However, it also comes at a cost. In this section, we look at the overhead of independent extensions composed at runtime versus a "monolithic" extension that combines their functionality.

Table 1. Overhead of extension composition on Excel startup while loading an 11kB input file. Mean and standard deviation (σ) over 100 runs are reported.

	Startup time s (σ)	Committed memory MB (σ)
No extensions	3.6 (0.3)	73 (14)
Tracing	3.6 (0.2)	74 (14)
FS translation	3.5 (0.3)	76 (20)
Both (monolithic)	3.5 (0.2)	79 (30)
Both (composed)	3.6 (0.2)	77 (8)

We use as our workload Excel with the 11 kB input file. Table 1 reports the startup time and committed memory for a variety of configurations: no extensions, only the tracing extension, only the file system translation extension, an alternative extension that we implemented specifically for this test which duplicates the functionality of both tracing and filesystem translation extensions in a single binary, and finally the two separate extensions composed at runtime.

We see that, much as in the previous experiments, the overhead of extensions is so low as to be within the noise. This confirms that the mechanism Bascule provides for composing extensions is of sufficiently low overhead that it should not be a concern when choosing to use extensions; rather, the performance behaviour of individual extensions will be the more important factor.

6. Discussion and future work

Implementation effort The effort required to implement the various components of Bascule varies. Clearly, the LibOS involves the most existing code and requires the most work. The cost of refactoring Windows 7 to produce a LibOS for Drawbridge was found to be tractable: 16k changes and 36k additions out of 5.6M lines of code total in the LibOS, completed in less than two person-years [29]. Our experience with Linux is only early, but indicates similar effort: it took one of us 12 weeks to construct the initial functional LibOS, but there is certainly much more work required to achieve full compatibility and optimise performance. Although each LibOS requires significant implementation effort, we expect few will be required, since there are correspondingly few dominant OS APIs.

Implementing a PAL for Bascule is comparatively less work, and requires mapping the ABI calls to the host platform. However, the real value of Bascule lies in its support for extensions, so we expect that there will ultimately be many more extensions than either LibOSes or host PALs. To write an extension requires understanding only the Bascule ABI, and we showed how simple extensions such as the tracer can be implemented in relatively little code. The programming environment for extensions is spartan, and simply consists of the ABI calls. This is both a drawback and benefit: since there is no runtime support extensions must pro-

vide their own, but conversely there is no complex internal OS interface with which extensions must remain compatible. We have begun to develop a set of self-contained support libraries for extensions using the Bascule ABI, however extension developers are always free to choose their own.

Multi-process support At present, the Bascule ABI includes functionality derived from Drawbridge to create child processes and communicate with them over pipes, however it is incapable of supporting general inter-process sharing, nor of implementing `fork`. Surprisingly few Windows applications use child processes (and the API has no fork primitive), but for Linux and other potential LibOSes this is insufficient.

We are evaluating two solutions for supporting multi-process applications, both of which entail expanding the confinement boundary between the LibOS and host OS to encompass multiple address spaces within a single sandbox. The first, more straightforward, approach would extend the ABI with new calls providing the ability to create and manipulate address spaces and their threads, and share resource handles between them. The main downside of this is a significant increase in the complexity of the ABI. Our second approach is motivated by the Dune system [3], and uses hardware support for virtualisation to run multiple "child" address spaces within a single container, while maintaining roughly the same ABI with the host (and with extensions). Both cases would require support within the host and LibOS, but the latter has the advantage that the management of child address spaces is entirely under the control of the LibOS.

Protecting extensions from applications Bascule fully protects the host OS and other applications against malicious applications and extensions, but does not attempt to protect extensions from the applications they extend. This enables lightweight extensibility by reducing the overhead of each extension, and keeps extensions outside the system's trusted computing base. However, it also precludes some possibly useful extensions that require protection from malicious (or buggy) application and LibOS code.

To maintain the safety of such extensions, they must be run in a separate (user-mode) address space. It is relatively straightforward to provide glue stubs that issue ABI calls as RPCs to a separate process where they are converted back to ABI calls, allowing such extensions to transparently use the same ABI. However, besides ABI call parameters, extensions may also assume access to the guest process address space. Absent hardware support for fine-grained sharing [35] the cleanest way to provide this for an unmodified extension would be to mirror the two address spaces, so that all memory visible to the guest is also visible to the protected extension, but not vice-versa. We are considering adding this support, but would only use it for specific extensions that require such protection (which we expect to be rare) or as a debugging aid.

7. Related Work

Bascule borrows ideas from many prior systems, including extensible operating systems, interposition mechanisms, and API compatibility libraries.

Past work on extensible OS architectures [4, 7, 14, 15, 31] generally focused on engineering new OSes from scratch to support fine-grained, secure extensibility. While the resulting systems were highly extensible, they often lacked compatibility with existing applications. For example, Exokernels [14] support OS extensibility by placing the majority of OS functionality in a user-mode library, permitting an application to supply its own customised LibOS. The main focus of this work was improving performance through application-specific optimisations to a LibOS [20]. While this is also possible in Bascule, we see the LibOS primarily as a compatibility mechanism, and provide a more limited form of extensibility via interposition on the Bascule ABI. Our intuition is that few application developers have the motivation to customise a LibOS. Moreover, a custom LibOS would be unable to receive security patches directly from the OS vendor. Rather, interposing at the Bascule ABI level enables extension development independently of a specific application or LibOS, and supports late-binding; extensions chosen by a user may provide functionality unforeseen by the application developer.

Interposition has also been used as an extensibility mechanism by prior systems. The Fluke microkernel [15] supports "nester" processes, that implement OS extensions such as checkpointing and demand paging. Each nester is implemented by a separate process, so keeping the overhead of nesters low requires the use of selective interposition, however common nesters such as the checkpointer interpose on all system interfaces. SLIC [16] targeted extensibility for a mainstream OS with in-kernel extensions that interpose on the Solaris system call interface. While this has low overhead, and supports extensions that perform additional security checks, extensions must be fully trusted. Bascule runs extensions in application context, which limits its use for security purposes, but at the same time protects the system from malicious extensions while retaining low overhead. We find that many extensions are suited to this trust model.

The Spring OS supports extensible file systems through stacking of extensions that add functionality such as data compression or distribution [21]. Spring's file and memory management interfaces are defined to enable efficient use of caches and file metadata across multiple extensions, and to permit interposition on a per-file basis. Bascule also enables extending the behaviour file system on a per-application basis (for example, by encrypting or compressing all file data transparently to the application), but is more general and thus lacks the filesystem-specific extension interfaces of Spring.

Many commodity operating systems also support an interposition mechanism for system calls intended for debugging purposes, such as ptrace in Unix-like systems, and it is

also possible to interpose on OS library APIs [18]. However, the complexity of the relevant interfaces makes these solutions demanding to use, and tightly couples the implementation of an extension to a specific OS. Moreover, composing extensions using such mechanisms is often impossible. Following the insight that despite the large number of system calls in an OS interface there are relatively few abstractions, interposition agents [19] provide an object-oriented extension API to simplify the task of interposing on system interfaces, while also supporting composition. However, interposition overheads are high, and extensions are still dependent on a specific OS interface. Motivated by a similar observation, the Bascule ABI was designed from the outset to use a small and stable set of abstractions, and naturally supports composition. Moreover, because interposition code runs in the context of the application, it has near-zero overhead.

API or ABI compatibility libraries [1, 11, 17, 34] allow code written for one OS to run on another, much as we support Linux and Windows applications on other host platforms. This is essentially a problem of interface adaptation, and LibOSes written to the Bascule ABI make use of many of the same techniques. The key difference is that the adaptation code is decoupled between the LibOS and host PAL – since the LibOS targets a common intermediate ABI, it can be reused by all Bascule host platforms.

Closely related systems to Bascule include Xax [12] and Drawbridge [29]. Xax enables the use of legacy code as secure browser plugins, and uses picoprocesses for confining untrusted native code, combined with an ABI for access to system services. Drawbridge showed how to refactor a large existing monolithic OS (Windows) to create a self-contained library OS capable of running in a picoprocess yet supporting rich desktop applications; its ABI is richer than the one used by Xax, consisting of a set of abstractions designed to mimic core Windows functionality. Our ABI is derived from it, which allowed us to port and reuse the Drawbridge LibOS, but it is more general. We also used similar techniques to develop the Linux LibOS described in §4.1. While Bascule retains all the benefits of Drawbridge, our focus in this work is on generality (across varying host and guest operating systems), and on supporting lightweight, composable OS extensions at runtime using interposition.

8. Conclusion

We presented Bascule, a LibOS extensibility architecture that allows application behaviour to be customised by extensions loaded at runtime. Bascule shows that it is possible to support lightweight yet safe OS extensibility as well as composition of extensions by running them in a shared address space and exploiting interposition on a common binary interface of primitive OS abstractions. We also showed that the use of an abstractional interface to the LibOS and extensions does not restrict Bascule to a single LibOS personality or host environment, but rather that a variety of OS personalities can be expressed above carefully-chosen abstractions.

Providing LibOS extensibility in the manner of Bascule is appealing, because it avoids the need to modify the library OS, allowing it to be patched and updated. Ultimately, we envision these techniques being used to support an "extension store" model where extensions are developed independently of both the OS and applications, and deployed at runtime by end users or system integrators.

Acknowledgments

We would like to thank the anonymous reviewers, Jon Howell, and particularly our shepherd, Frans Kaashoek, for their constructive feedback that significantly improved this paper.

References

[1] Jonathan Appavoo, Marc Auslander, David Edelsohn, Dilma Da Silva, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. Providing a Linux API on the scalable K42 kernel. In *Proceedings of the 2003 USENIX Annual Technical Conference, FREENIX Track*, pages 323–336, June 2003.

[2] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 29–44, October 2009.

[3] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, pages 335–348, October 2012.

[4] Brian N. Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, December 1995.

[5] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, August 2003.

[6] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, February 1996.

[7] David R. Cheriton and Kenneth J. Duda. A caching model of operating system kernel functionality. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, 1994.

[8] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation*, pages 273–286, 2005.

[9] Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 337–351, 2011.

[10] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: high availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174, 2008.

[11] Cygwin. http://cygwin.com/.

[12] John R. Douceur, Jeremy Elson, Jon Howell, and Jacob R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 339–354, December 2008.

[13] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, pages 211–224, 2002.

[14] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, December 1995.

[15] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 137–151, October 1996.

[16] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, and Thomas E. Anderson. SLIC: An extensibility system for commodity operating systems. In *Proceedings of the 1998 USENIX Annual Technical Conference*, June 1998.

[17] David Given. LBW: Linux binaries on Windows, April 2010. http://lbw.sf.net/.

[18] Galen Hunt and Doug Brubacher. Detours: Binary interception of Win32 functions. In *Proceedings of the 3rd USENIX Windows NT Workshop*, July 1999.

[19] Michael B. Jones. Interposition agents: transparently interposing user code at the system interface. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 80–93, 1993.

[20] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on Exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 52–65, October 1997.

[21] Yousef A. Khalidi and Michael N. Nelson. Extensible file systems in Spring. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 1–14, 1993.

[22] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 1–15, April 2005.

[23] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. Respec: efficient online multiprocessor replay via speculation and external determinism. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 77–90, 2010.

[24] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7), September 1996.

[25] Microsoft. *Programming Guide for 64-bit Windows: Running 32-bit Applications*, June 2012. http://msdn.microsoft.com/library/windows/desktop/aa384249.aspx.

[26] Dejan S. Milojičić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Computing Surveys*, 32(3):241–299, September 2000.

[27] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative execution in a distributed file system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 191–205, 2005.

[28] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[29] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinksy, and Galen C. Hunt. Rethinking the library OS from the top down. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–304, March 2011.

[30] Timothy Roscoe, Kevin Elphinstone, and Gernot Heiser. Hype and Virtue. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*, May 2007.

[31] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 213–228, November 1996.

[32] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. Triage: diagnosing production run failures at the user's site. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 131–144, 2007.

[33] Andrew Whitaker, Richard S. Cox, Marianne Shaw, and Steven D. Gribble. Constructing services with interposable virtual hardware. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation*, pages 169–182, 2004.

[34] Wine. http://www.winehq.org/.

[35] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 304–316, 2002.