How to Run POSIX Apps in a Minimal Picoprocess

Jon Howell, Bryan Parno, John R. Douceur
Microsoft Research, Redmond, WA

Abstract

We envision a future where Web, mobile, and desktop applications are delivered as isolated, complete software stacks to a minimal, secure client host. This shift imbues app vendors with full autonomy to maintain their apps' integrity. Achieving this goal requires shifting complex behavior out of the client platform and into the vendors' isolated apps. We ported rich, interactive POSIX apps, such as Gimp and Inkscape, to a spartan host platform. We describe this effort in sufficient detail to support reproducibility.

1 Introduction

Numerous academic systems [5, 11, 13, 15, 19, 22, 25–28, 31] and deployed systems [1–3, 23] have started pushing towards a world in which Web, mobile, and desktop applications are strongly isolated by the client kernel. A common theme in this work is that guaranteeing strong isolation requires simplifying the client, since complexity tends to breed vulnerability.

Complexity evicted from the client kernel takes up residence in the apps themselves. This shift is beneficial: It lets each app vendor decide independently which complexity is worth the risk of vulnerability, and one vendor's decision in favor of complexity does not undermine another's decision to favor security. Of course, requiring each app vendor to implement a complete software stack is impractical, so we expect this complexity to migrate to app frameworks that app vendors can choose among, just as web developers choose among an ever evolving set of app frameworks on the server.

The minimality of the client interface must not inhibit the richness required by applications such as desktop productivity apps. New client application models often fail due to the burden of migrating every app–and every library–to run under a new model. Thus, we argue that shifting app delivery to a minimal-client model requires an inexpensive app migration path from complex-host frameworks such as POSIX and Windows.

On the other hand, support for richness should not sacrifice the small size and tight specification of the isolation interface. The web's current client execution interface has repeatedly failed to achieve strong app isolation, due to an interface bloated with HTML, DOM, JPG, PNG, JavaScript, Flash, etc. in pursuit of richness.

The recent Embassies system provides a concrete example of how to achieve both security and richness simultaneously [16]. It pushes the minimal client host interface to an extreme, proposing a client host without TCP, a file system or even storage, and with a UI constrained to simple pixel blitting (i.e., copying pixel arrays to the screen). In support of rich apps, Embassies's minimal interface specifies execution of native binary code. Native code is an important practical choice, because, we assert, it is the lack of native code that has forced each prior system based on language safety to evolve a complex trusted interface that provides access to native libraries [8, 10, 17, 20]. This complexity undermines the intent to provide strong security.

		Libraries	
Application	**Function**	**#**	**Examples**
Abiword	word processor	63	Pango,Freetype
Gimp	raster graphics	55	Gtk,Gdk
Gnucash	personal finances	101	Gnome,Enchant
Gnumeric	spreadsheet	54	Gtk,Gdk
Hyperoid	video game	6	svgalib
Inkscape	vector drawing	96	Magick,Gnome
Marble	3D globe	73	KDE, Qt
Midori	HTML/JS renderer	74	webkit

Table 1: *A variety of rich, functional apps transplanted to run in a minimal native picoprocess. While these apps are nearly fully functional, plugins that depend on* `fork()` *are not yet supported (§3.9).*

While native code is a target that every compiler can hit, it seems daunting to port arbitrary POSIX apps to such a minimal interface. Such apps expect to run on a complex host with hundreds of system calls and dozens of system services, reflecting decades of development.

However, our experience suggests this task is far easier than one might expect. Interactive apps use relatively little of the complexity available in modern host platforms. More importantly, rather than alter the app, the functions that are required can often be emulated behind the POSIX interface. This technique works without even recompiling the hundreds of libraries involved. The emulation work can be shared easily across many applications, making the porting work scalable. The broad selection of rich apps that our system supports (see Table 1) demonstrates the generality of the approach.

Contributions. This paper demonstrates the tractability of porting rich POSIX apps to a minimal environment, thus enabling them to run on a multitude of minimal client hosts [13, 16, 18, 22, 31]. We give a full account-

ing of the porting task, including which functionality is required and where corners can be cut. This includes low-level details, such as an exhaustive list of syscalls handled, to enable reproducibility and to eliminate any ambiguity about complexity hidden under the hood. Ultimately, we hope that this will expedite other efforts to adopt these techniques and hence achieve rich applications atop minimal, strongly-isolating client kernels.

2 Background: Minimal Client Facilities

In this work, we aim to transplant apps from a rich POSIX interface to a minimal client kernel. To ground the discussion, we target the minimal Embassies *picoprocess* interface [16], since it takes minimality to an extreme. If we can port an app to Embassies, we can certainly port it to a client with a richer interface.

The Embassies application binary interface (ABI) provides *execution* primitives that support an app's internal computation, *cryptographic* primitives to facilitate privacy and integrity, primitives for IPC and network *communication*, and *user interface* (UI) primitives for user interaction.

Execution. The execution primitives include:

- Calls to `allocate_memory` and `free_memory`. To simplify the specification and to make the ABI portable to most host environments, the app specifies only the amount of memory required; it has no control over the addresses returned by the allocator.
- `create_thread` accepts only the thread's initial program counter and stack pointer; the application provides the stack and any execution context.
- `exit_thread` destroys the current thread.
- A simplified futex-like [6] synchronization scheduling primitive, the *zutex*. `zutex_wake` is a race-free scheduling primitive that supports app-level efficient synchronization primitives. The corresponding `zutex_wait` is the only blocking call in the ABI; it allows an app to yield the processor.
- `clock` returns a rough notion of wall-clock time.
- `set_timer` sets a timer, in clock coordinates, that wakes a zutex on its expiration. Each picoprocess has only one timer; the app must multiplex it.
- `get_alarms` returns a list of three distinguished zutexes representing external events, one for each of `receive_packet`, `ui_event`, and `timer_expired`. Waiting on these zutexes is how threads block on external activity.
- A call to create a new picoprocess.

Cryptographic Infrastructure.

- `random` provides a supply of cryptographically strong entropy.
- `app_key` provides a machine-specific, application-specific secret. Apps use this key, along with cryptographic libraries, to store and recover private information despite starting from a public binary.

Communication. All communication outside the process, whether IPC to another process on the local machine, or remote to an Internet host, follows IP semantics: Data is transferred by value (a logical copy), so that the suspicious recipient needn't worry about concurrent modification; addressing is non-authoritative; delivery admits loss and duplication; packet privacy and integrity are not guaranteed. Just like servers on the Internet, apps build up integrity and privacy themselves using cryptography. To underscore these semantics, all communication in Embassies–remote or local–is done via IP.

- `get_addresses` assigns the process one IPv4 and one IPv6 address.
- `allocate_packet` allocates memory for an outgoing packet; this allocation is distinguished from `allocate_memory` to enable zero-copy transfer.
- `send_packet` delivers a packet, interpreting its argument as an IP header and payload.
- `receive_packet` returns an allocated and dequeued packet, or NULL if the queue is empty.
- `free_packet` frees an allocated packet.

User Interface.

- `ui_event` returns a dequeued UI event (keystroke or pointer motion), or NULL if the queue is empty.
- Some calls that manage viewports, letting them be transferred among applications, or letting one application sublet a region of its viewport to another application. In every case, even where nested, each viewport is owned by a single app; no app can inspect or modify the pixels of another app's viewport. Details can be found elsewhere [16].
- `map_canvas` allocates a framebuffer to back a viewport. This allocation is distinguished from `allocate_memory` to enable fast pixel blitting.
- `update_canvas` informs the client kernel that a region of the framebuffer has been updated, and that its pixels should be blitted to the display.

These calls comprise the entire Embassies ABI; all of the functionality described in the rest of the paper is implemented in terms of these primitives.

3 The POSIX Emulator

A conventional POSIX application employs dozens of libraries, access to a rich system call interface, and by way of those system calls, access to other rich services, such as the `X` server's graphics functions and the `dbus` desktop configuration object broker.

To execute applications expecting this rich POSIX environment, our POSIX emulator cleverly repurposes existing libraries and programs atop the execution environ-

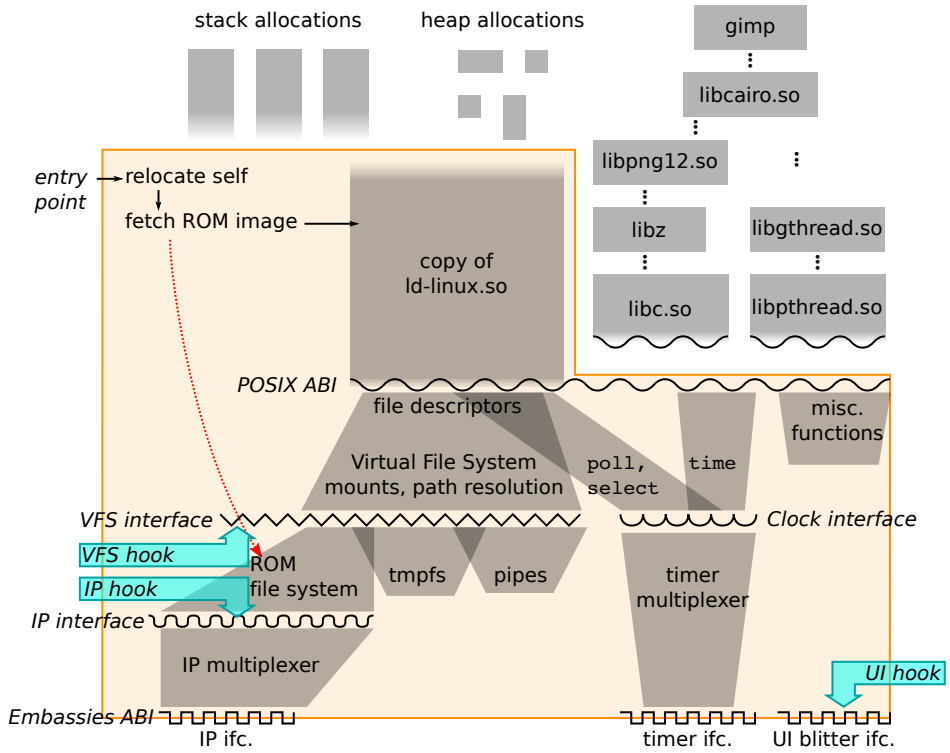

Figure 1: **The POSIX Emulator.** *To Embassies, the emulator (the large, L-shaped boundary) is a binary string whose entry point is its first byte, and which may call back into a set of low-level interfaces provided by the Embassies ABI. Internally, the emulator loads the app's read-only image, maps it into a virtual filesystem, and calls into a copy of* `ld-linux.so`*. That loader, using the emulated POSIX ABI, reads the app executable and additional ELF libraries into memory. The* `glibc` *libraries' syscalls are redirected to the emulator's POSIX interface. Non-POSIX hooks provide connections for UI and TCP services implemented outside of the emulator (Figure 2).*

ment's minimal services (§2). Figure 1 gives a structural overview of how the emulator maps the entire POSIX interface down to Embassies's picoprocess interface.

Below, we provide a functional exposition of this emulation, starting with application launch.

3.1 Application Launch

Embassies provides minimal support for app launch, merely loading and starting a vendor-specified *boot block* of code. Specifically, the host (1) maps the applications' boot block into an arbitrary region of address space, (2) sets up a minimal stack, and (3) places in a register the address of a dispatch table for the Embassies ABI (§2).

Within its boot block, the POSIX emulator (1) relocates its symbols, using a small piece of position-independent code, (2) allocates an adequate stack, and (3) establishes a dispatch function to emulate the POSIX syscall interface (§3.2) and virtual file system (§3.3).

Next, the emulator must load the app and its libraries into memory. In a full Linux implementation, the kernel would interpret the app's ELF binary format, map the app binary into memory, map the loader `ld-linux.so` into memory, and then jump to the loader. The loader would then enumerate dynamic library references within the ELF image, map these libraries into memory, link the images together (resolving symbolic references), and then jump to the app's entry point.

Embassies, however, provides neither a file system from which to map files nor a kernel willing to parse ELF binaries. Thus, our emulator must perform these tasks, which it does by invoking `ld-linux.so`, an image of which is included in the emulator's boot block. The emulator calls `ld-linux.so` and passes the app's path as an argument, which instructs the loader to map the app (and its libraries) into memory. POSIX calls made by `ld-linux.so` are serviced by the emulator (§3.2).

To call the loader, the emulator creates a suitable `argv` (naming the ELF executable), an `envp` (e.g. pointing `DISPLAY` at 127.0.0.1:6), and an `auxv` (some constants to convince libraries they're running on Linux).

3.2 Intercepting System Calls

The loader, as well as other libraries in the glibc suite, are at the bottom of the library stack; these are the libraries that make actual POSIX syscalls. In principle, other libraries could also include direct syscall instructions, but in practice, we have never observed this; instead, they simply use `libc`'s `syscall` symbol.

We want to exploit the functionality of the glibc suite, but glibc's system calls will fail in an Embassies process; they must be intercepted and replaced with calls to the syscall emulation layer. In principle this can be achieved by creating an alternate "sysdep" personality for glibc. In practice, at least for the x86 architecture, we found it easiest to apply a binary rewriting pass to each of the libraries in the glibc suite, patching every system call invocation (i.e., each occurrence of `int $0x80`) with a call to a dispatch function that we inserted at the end of the library.

The dispatch function in each library must, in turn, be patched dynamically to call into the emulator's syscall dispatcher. To identify libraries in need of such dynamic patching, we modified the libraries' ELF headers to label the dispatch function. As libraries are `mmap`ed into the app's address space, a filter file system in the VFS layer (§3.3) detects the modified ELF signature and transparently updates the dispatch function to point at the emulator's syscall dispatcher.

3.3 Virtual File System

Much of the POSIX ABI concerns file naming and file descriptors, which provide access to a variety of functions. Thus, like a Unix POSIX implementation, the emulator contains a virtual file system (VFS) abstraction.

VFS components include a read-only app image, a RAM-based writable temporary filesystem (`tmpfs`) that implements POSIX scratch directories like `/tmp`, and named pipes (Unix-domain `sockets`). The writable `tmpfs` directories provide the namespace for the Unix-domain sockets. There are also the virtual files that emulate POSIX special files. These comprise the `/proc` files of Section 3.8.1 and an emulated `/dev/random` which passes entropy up from the client kernel's `random` facility.

The emulated VFS contains an overlay mount table to weave these file systems together.

3.3.1 The Read-Only Application Image

The most important VFS component is the read-only binary image, whence libraries and data files are fetched.

A Linux app expects to fetch its libraries and read-only data files by name from a (shared) file system via `read` and `mmap`. In Embassies, such files come from a private app image whose integrity has been verified.

To support this, the developer packages every file the app requires into a single tar-style image file. The emulator fetches this file from an untrusted cache on the local machine, delegating to the cache the complexity of fetching the image from an upstream cache or origin server and exploiting commonality with other apps [14]. The reply appears in memory as a single (jumbo) IP packet. The emulator ensures integrity by comparing the image's hash to a fixed hash value embedded in the boot block.

`accept`	`recvfrom`
`bind`	`recvmsg`
`connect`	`send`
`getpeername`	`sendmsg`
`getsockname`	`sendto`
`getsockopt`	`setsockopt`
`listen`	`shutdown`
`recv`	`socket`

Table 2: **Socket Calls.** These calls are plumbed through the VFS interface to either the Unix named pipes implementation or the TCP stack.

The image file transmission protocol supports partial fetches, so that the app can start with only a subset of the image, and then later page in additional components.

3.3.2 Supported Interfaces

POSIX defines a wide, complex interface for interacting with the file system, so implementing the entire interface would be quite labor intensive. Fortunately, to support the varied applications from Table 1, it suffices for the VFS to support the following functions.

First, there is the core interface `open`, `close`, `ftruncate`, and `ftruncate64`; and the metadata interface `stat`, `lstat`, `fstat`, and `access`. VFS file descriptors track file pointers for `read`, `write`, `writev`, and `lseek`. Directory functions `mkdir`, `getdents`, `getdents64`, (hard) `link`, and `unlink` are only implemented in the `tmpfs`. The socket calls (Table 2) are routed through the VFS to the Unix pipe and TCP (§4.2) implementations.

The emulator also implements file handle functions `dup`, `dup2`, `pipe`, and `pipe2`. `pipe` connects two file descriptors with a blocking pipe with no presence in the VFS namespace. Functions `fsync` and `fdatasync` are no-ops. Most of `fcntl` and `fcntl64` are no-ops, except `F_DUPFD`, which calls the `dup` implementation.

3.4 Mmap Support

POSIX `mmap` is versatile, but in practice it is used in only a few idiomatic ways.

First, `mmap(MAP_ANONYMOUS)` is used to allocate blank memory at an address chosen by the kernel. The emulator transforms these calls into Embassies memory allocations.

Second, apps use `mmap` explicitly to map in non-executable data files. These calls also give the emulator freedom to choose the target address, so the emulator allocates fresh memory and uses a `memcpy`ing `read` implementation to simulate the effect of the `mmap`.

Finally, apps use `mmap` implicitly when they dynamically link executable libraries, either at load time via `ld-linux.so` or at runtime via `dlopen`. Some of these calls *do* expect to control the resulting data placement, a degree of control that Embassies does not provide when allocating memory.

Fortunately, the loader does not *really* care where a given library ends up; it just requires that the data segment of the library appears at the correct offset from the text segment. To this end, the loader's first `mmap` call does not specify a target address; instead, it specifies a length sufficient to reserve enough address space to cover all the segments in the file. The loader's subsequent `mmap` calls (e.g., for the data segment) do specify a target address, but the target address is always within the memory range allocated by the initial `mmap` call.

Thus, the emulator can support this final class of `mmap` calls by simply using the Embassies interface to allocate the initial memory region (which does not specify a particular address), and then confirming that subsequent `mmap`s (that do specify an address) fall within the initial memory allocation. As long as they do, the emulator can take the appropriate action, e.g., it can zero-fill the specified region for the binary's `.bss` section or copy in the contents of the `mmap`ed file.

This approach is clearly "less portable", in the sense that a POSIX app could in theory call `mmap` with an address outside of any preexisting allocation. Fortunately, we have not yet encountered any applications that rely on this functionality.

3.4.1 Fast `mmap`

The approach above is adequate for correct POSIX emulation, but for the apps we tested, where the bulk of the image comprises `mmap`-loaded libraries, it incurs many megabytes of `memcpy`s, adding noticeable delay (150 ms) to the app start time. We corrected this performance problem by page-aligning `mmap`able libraries in the image tar file (§3.3.1), and servicing `mmap` requests by yielding the memory region from the VFS to the app.

Of course, this means that the region can not be `read` or `mmap`ed later in the program's execution; if a program needs to map a file multiple times, we either store multiple copies in the image file (often worth the space), or mark the region "precious", inhibiting the optimization.

Fast-`mmap` files must be stored in the image in their in-memory layout, not their on-disk ELF layout, including necessary blank space to position the data and bss segments. The blank spaces are, of course, easy to compress during transmission.

3.4.2 Other Memory Calls

Most POSIX memory allocations appear as anonymous `mmap` calls. The emulator tracks such requested regions, freeing the underlying Embassies allocation once the entire region has been `munmap`ped.

Embassies provides no read/write/execute memory protections, so the emulator simply ignores `mprotect`, `madvise`, and `msync`. It also rejects `mremap`.

Unfortunately, `ld-linux.so` and `libc` both make initial memory allocations with the ancient `brk` interface. Why? We cannot say; the best solution would be to eradicate these deprecated calls. Instead, as a workaround, the emulator assumes that virtual memory has no cost, generously over-allocates on the initial `brk(0)` call, and services each subsequent `brk` extension by releasing more of the initial allocation.

3.5 Clock and Timers

The emulator provides the various flavors of POSIX time: `time`, `gettimeofday`, and `clock_gettime`. It translates all of these from the nanosecond precision clock supplied by the client kernel. That clock provides rate but no offset information; hence all of our apps think the current time is 2011. We use ntpdate to acquire a clock offset, although we have not yet attended to the security implications.

Embassies supplies the process with a single timer, which signals the process by firing a zutex, and thus can be reset in a race-free way. The emulator has the responsibility to multiplex this one timer into as many alarms as it needs to implement POSIX timeout interfaces. It does so using a tree of upcoming deadlines, for scalability. We found the clock multiplexer to be surprisingly subtle, with many race conditions that lead to deadlocks. It was helpful to diagram the detailed mapping between the host timer state and the state of the guest timer list.

3.6 Synchronization Primitives

The Embassies client kernel provides a single unified synchronization abstraction, the `zutex`, that is used both for internal waiting on other threads and waiting on external events (the network or the clock). This central abstraction is a simplified `futex` [6]. Like the futex, the zutex is actually a race-free *scheduling* primitive in support of efficient synchronization.

The basic POSIX `futex` maps readily onto the zutex, with the emulator folding in timeout behavior (§3.5). Many extra POSIX behaviors are neutered. For example highly concurrent servers use FUTEX_CMP_REQUEUE to avoid convoys, but our emulator simply wakes the requested threads and lets them requeue themselves. The emulator rejects FUTEX_WAKE_OP and FUTEX_WAIT_BITSET with an error, alerting `libpthread` to revert to the basic behavior.

The `nanosleep` call and POSIX multiple-wait primitives `select`, `newselect`, and `poll` are all mapped into `zutex_wait` operations, again with timeout behavior constructed by the emulator. POSIX blocking operations, like a `read` on an empty pipe, wait on zutex-signaled events.

3.7 Network Multiplexing

Embassies provides each process with a single `zutex` to signal the arrival of IP traffic. Thus, the emulator must collect incoming IP packets and multiplex them

inside the app. The emulator itself uses IP to fetch its image (§3.3) and for querying time servers (§3.5). The emulator's network stack demultiplexes IP and UDP, and delivers TCP packets to the LWIP library (§4.2).

3.8 Threads

POSIX uses `clone` to express both thread creation and process fork (§3.9). The emulator pattern-matches the thread-creation idiom and sets up the new thread's initial thread-local store (TLS). Because Embassies' `create_thread` conveys only a stack pointer, the emulator constructs a stub stack to pass the POSIX parameters and the caller's designated stack to the new thread. It records metadata about the new thread to correctly implement `CLONE_CHILD_CLEARTID` upon POSIX's thread-exit call.

The POSIX process-exit call, `exit_group`, signals the zone host (§4) that a zone has exited.

3.8.1 Supplying the Stack Address

Several applications rely on garbage collection libraries that need to know the address of the top (but not the bottom) of the current thread's stack. This is exposed in Linux POSIX through pseudofiles in `/proc`.

At first blush, it appears that the stack bottom address is also needed. For example, Libwebkit's JavaScriptCore garbage collector queries `libpthreads` for the stack bottom address. However, the GC does not use the bottom address directly; instead, it adds `RLIMIT_STACK` to yield the top address. Since `libpthreads` determines the bottom address by subtracting `RLIMIT_STACK` from the top address it obtains from `/proc/self/maps`, any sane value for `RLIMIT_STACK` will work correctly. We used 8 MB.

The stack top value returned by `/proc/self/maps`, on the other hand, does matter: It is how a conservative garbage collector learns the extent of the stack. Another garbage collector, `libgc`, looks for the stack top in `/proc/stat/self`. We install special VFS nodes at those names which return the appropriate stack top value for the current thread.

To identify which thread is querying the interface, the emulator snoops the app's thread-local store (TLS) register; that is, it uses grey-box assumptions about how glibc manages the TLS. For all threads other than the main thread, the emulator records each stack address as its thread is created by the `clone` syscall. For the main thread, the emulator allocated the stack (§3.1) and thus knows its address.

3.9 Unimplemented: Fork

Some apps employ the `fork/exec` pattern; e.g., Inkscape uses it for its plug-in modules. This pattern does not translate well to the minimal Embassies environment, since Embassies's memory management facilities are far too simple. The current emulator implementation does not support `fork` at all, leaving Inkscape's plug-ins inoperative.

An expedient approach, if the code is sufficiently idiomatic, is to emulate the fork with a thread, and perhaps intercept and neuter `close` calls from the child "process" preparing to `exec`. The `exec` call would launch a new zone (§4), or if fault-containment is desired, a new picoprocess (§2).

Alternatively, since the `fork/exec` pattern is usually implemented in a widely-used library, such as glib's `g_spawn`, one could modify this higher-level library to map fork's semantics cleanly onto the creation of a new zone or picoprocess.

`chmod`	`sched_setparam`
`chown`	`sigaction`
`fchmod`	`sigprocmask`
`rename`	`umask`
`sched_get_priority_max`	`sched_setscheduler`
`sched_get_priority_min`	

Table 3: **Failure-oblivious calls** return either EINVAL or ENOSYS, which the caller handles gracefully.

`fchown`	`set_tid_address`
`flock`	`setitimer`
`fstatfs`	`setpriority`
`inotify_init`	`setrlimit`
`inotify_init1`	`shmget`
`ioctl`	`statfs`
`ipc`	`sysfs`
`readlink`	`times`
`sched_getaffinity`	`xi_sched_yield`
`set_robust_list`	`xi_timer_create`
`xi_sched_rr_get_interval`	

Table 4: **Neutered calls** simply return 0 (success).

3.10 Neutered System Calls

The remaining syscalls are either unused by interactive apps, or can be simply rejected or neutered. This section identifies such syscalls in the interest of completeness.

Many calls (Table 3) can be rejected, returning `ENOSYS` or `EINVAL`, and the libraries that call them either handle the failure gracefully, fall back to an alternate POSIX mechanism, or ignore the result and trundle along obliviously [24].

Other syscalls can be neutered with brazen lies: When the caller actually checks the return code, we may need to return 0 ("success") even if we don't actually emulate the promised semantics (Table 4). Other functions require slightly more credible lies: The emulator fills in some plausible constant values to placate the caller (Table 5). For instance, the `clock_getres` call should provide some information about clock quality (§3.5), but we just claim a 500 ms resolution. As another example, we found no software that used `chdir`, so `getcwd` simply returns "/".

clock_getres	getpid
getcwd	getppid
getegid	getresgid32
getegid32	getresuid32
geteuid	getrusage
geteuid32	getuid
getgid	getuid32
getgid32	sched_getparam
getpgrp	uname
sched_getscheduler	

Table 5: **Deluded calls** return slightly fancier lies than 0.

3.11 Additional Program Requirements

Emulating the POSIX ABI is minimally intrusive to the apps, but a few conflicts remain.

3.11.1 Address Freedom

We have already seen that Embassies's refusal to let apps specify specific locations for allocated memory requires the boot block to relocate itself (§3.1) and requires a delicate hand in servicing `mmap` (§3.4).

It also means that every executable must be relocatable or position independent. Every Linux shared library is relocatable, but for no discernible reason, executables are not relocatable by default. We address this by rebuilding each app's top-level executable with the `-pie` ("position-independent executable") compiler flag. Although this requires tampering with the app's build system (§6.1), it is required only for the top-level application, not any libraries; and in most cases, passing `DEB_CFLAGS=-pie` to `dpkg-buildpackage` does the job. The change is nowhere near as invasive as trying to change to static linkage (§6.1).

3.11.2 The TLS Register

On the architecture we experimented on, the arcane x86-32 instruction set architecture, a paucity of general-purpose registers leads POSIX compilers to employ a disused segment register `%gs` as a thread-local storage (TLS) pointer. This usage gets compiled into every library and application binary. Since the idiom has no security-sensitive semantics, we opted to provide a `store-gs` call in the x86-32 Embassies ABI; the emulator uses it to implement `set_thread_area`.

A better solution would be to either recompile or binary rewrite every binary to eliminate `%gs` references.

4 Zones: Programs as Libraries

Besides kernel services, POSIX apps often expect access to higher-level services provided by daemon programs like `X` windows, a window manager (e.g., `twm`), or a configuration manager, like the `dbus` desktop bus. We satisfy such apps by including these services *inside* the apps that need them, rather than in the client kernel's TCB (which would add them to every app's TCB).

`X`, `twm`, and `dbus` are designed as independent

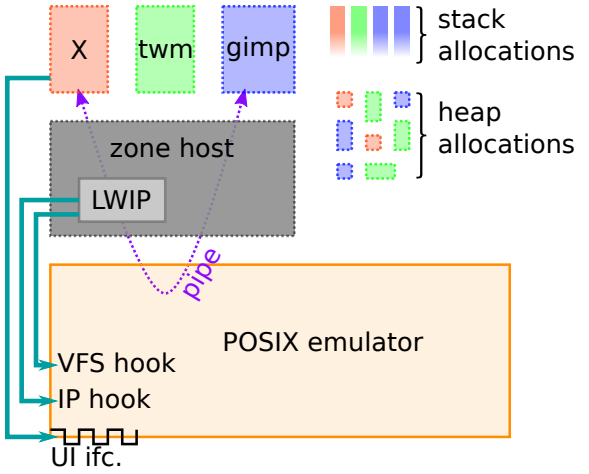

Figure 2: **Multiple POSIX apps coexist in one picoprocess as *zones*.** *Each zone comprises a noncontiguous partition of the address space. Each has its own copies of libraries, like `libc`, and its own stack and heap allocations. Programs that expect POSIX pipe IPC, such as an `X` session, see the same behavior within the picoprocess.*

POSIX processes. Rather than convert them into libraries, we found it expedient to create a general mechanism for loading multiple programs into a single picoprocess. This is easier than it sounds, because each program separately allocates memory and file descriptors, which carves the resource namespaces into interleaving partitions. We call such partitions "zones" (Figure 2).

Embassies's refusal to allow memory allocations at specific addresses works to our advantage when implementing zones, since it precludes zones from demanding overlapping allocations. It is zones that use the emulated Unix pipes (§3.3). For example, the `X` zone listens on `/tmp/.X11unix/X0`, and the xlib client library in the main application zone binds to it there.

The vestigial `brk` interface (§3.4.2), however, presents a hurdle. Two threads in different zones may concurrently extend different `brk` heaps. The `brk` interface assumes hidden per-process state, which becomes per-zone state. The good news is that we can infer which zone is making the request, and hence which per-zone state to consult, because each request should appear within the address space set aside for that zone's brk.

The bad news is that, on 32-bit hardware, virtual address space is scarce enough to warrant preserving, which means allocating only appropriately-sized `brk` regions for each zone. This is tricky because the initial call from each zone is a stateless `brk(0)`, from which the emulator cannot infer the identity of the calling zone. Our expedient solution forces the zones to start up sequentially. A more elegant solution would identify the calling zone by its TLS or stack pointer, or (better yet) eliminate `brk` calls from libc.

4.1 Example Zones

The X server presents a complex security boundary, and this complexity conflicts with Embassies's goal of a minimal client kernel TCB. Therefore we use X only inside the picoprocess, in a zone, disregarding its security-sensitive multiplexing functions and exploiting only its rasterization function. The rendered frame buffer that X produces is blitted to the user's display through Embassies's pixel-level UI interface.

Some apps, like Gimp, use a plethora of palette windows. For expediency, we add a twm window manager zone into such apps, to allow manipulation of the palettes within the surface of the app's single display region. With more effort, one could coordinate multiple windows via Embassies's window management, perhaps using a technique like Nitpicker's [9].

Gnome desktop apps expect to connect to the dbus daemon to find other components and learn configuration settings. This tight coupling among applications has no cost in a trusted-everything system, but is too risky for mutually untrusting apps. Hence we do not reproduce the connected dbus; instead, we link a copy of the daemon into each app to expediently satisfy the client library. With more effort, one could strip the dbus dependencies out of each app.

4.2 Extension Hooks

The emulator sits below libc, and hence cannot exploit libc. Coding without libc is painful; thus where possible, we push functionality out of the emulator into layers above. To facilitate this modularity, the emulator exports four hooks via unused syscall numbers.

Specifically, as alluded to above, we use an X zone to translate app UIs into easily blitted pixel regions. A modified X server within the zone supplies the graphical user interface. It uses one extension hook, ex_get_dispatch_table, to gain access to the raw Embassies UI functions. It uses a second extension hook, ex_open_zutex_as_fd, to wrap the UI notification zutex in a POSIX file descriptor, enabling the extension to smoothly integrate into X's existing poll loop.

All unhandled IP traffic, including TCP traffic, is handed off to a TCP stack based on lwIP [7] that resides in the zone host. The lwIP stack is a loadable module, attaching to the emulator's IP multiplexer via ex_add_default_handler and servicing requests for SOCK_STREAM sockets via ex_mount_vfs.

5 Debugging Strategies

The key premise of this work is that most apps use only a fraction of POSIX functionality. This paper catalogs these functions in detail precisely because the challenge is in discovering *which* functions matter.

Most of the effort in emulating the right subset of POSIX involves figuring out why a segfault occurred in a library dozens of layers below the app. To assist the practitioner who wishes to extend this approach, this section identifies our most valuable debugging strategies.

It is important to plumb error messages out of the picoprocess. Our insecure debug-mode Embassies monitor offers an extended ABI with debug channels that record to files. The emulator routes stdout and stderr to them.

Since most of our changes occur behind the POSIX interface, it is very effective to compare system call traces; divergences often identify root causes. We capture a reference trace in Linux with strace, and add a corresponding debug facility at the emulator's entry point. It emits a trace file using another debug output channel.

Of course, a debugger is invaluable. Our debug-mode monitor runs apps as Linux processes. It routes Embassies syscalls out through a pipe to a coordinating process, but leaves the conventional POSIX syscall interface intact, enabling gdb to connect to the process.

However, gdb has no access to symbols. The emulator does not use POSIX mmap to map in ELF files, so gdb's inspection of Linux-provided metadata in /proc/*pid*/maps is fruitless. To bridge this gap, the emulator records a trace of file open and mmap operations via another debug channel. A script transforms the trace into a gdb add-symbol-file script, solving the symbol problem.

Similarly, gdb's usual mechanism for discovering new threads fails when thread creation is handled by the emulator. Thus, the debug monitor provides another extension by which the emulator signals thread creation, and the debug monitor generates the appropriate trap (int $0x3) to alert gdb.

We haven't yet implemented gdb stubs for our secure monitors, because once an app runs correctly in the debug monitor, it rarely fails in the secure monitors. In the rare failure cases, we found it sufficient to study a core file (a snapshot at the moment of failure). Each secure monitor has a debug mode in which a picoprocess exception generates an ELF-format core dump.

The debug monitor also provides an extension to query CPU time (POSIX times()), and a sampling profiler, for diagnosing performance problems. An example discovery was that the emulator was returning bogus stat values, causing a font library to deem its cache file invalid, causing it to re-scan thousands of individual font files at app start.

Finally, gathering the appropriate file set for the read-only app image is tedious. To expedite, the emulator can start in "gullible mode", where rather than fetch an image, it passes every open request path out to a lookup server located on the development machine where the original POSIX app is installed. That server hashes the corresponding file, injects the file contents into the cache,

and returns the path to the emulator. By this means, the emulator demand-loads the app's required files; it also captures a trace of these loads, which serves as a manifest for generating the app image.

6 Discussion

Our goal is to reuse conventional interactive desktop applications in a new minimal runtime environment. Ideal reuse would use unmodified binaries; required modifications can be ranked based on their invasiveness.

Transparently emulating required behavior below the POSIX interface has proven to be very inexpensive; the main cost is discovering which features actually warrant implementation (§5). Our experience suggests that the emulator is asymptotically nearing completeness.

The choice to give apps no control over their memory layout, which makes Embassies implementable on any host, is slightly invasive; it requires relinking the top-level app binary, which is easy in practice (§3.11.1).

The Embassies environment demands some point changes higher in the software stack, including the binding of X to the Embassies UI interface (§4.1) and the replacement of implicit kernel communication with explicit protocols (§6.3). Such changes do require source modification of specific packages, but very few such changes are required, compared with the hundreds of library packages ported.

The Embassies impositions do preclude running some unmodified binaries, such as closed-source apps. Closed-source libraries with Embassies-compatible semantics, such as a PDF-rendering library, may be usable, though.

6.1 Dynamic vs. Static Linking

In our previous experience with the Xax project [13], we found that modifying a package's build system was frustratingly difficult, generally much harder than modifying the source code and using the package's build system to remake it. Most packages use common source languages such as C or C++, but it seems every package uses a different build scheme.

Engineering choices in Xax required statically linking each app with all of its libraries. Because that changed how apps and libraries build, the task ranged from difficult to all-but-impossible, and required new work for nearly every package.

Thus, in the present work, we elected instead to keep applications dynamically linked, and to press `ld-linux.so` into service for runtime linking. We found that this expedient substantially reduces the invasiveness of porting, as essentially every intermediate library is readily usable in binary form.

6.2 Limitations

Our experience iterating the emulator to support several apps suggests that the emulator is asymptotically nearing completeness, ready to support most desktop productivity apps. In most cases where we have introduced a lie or neutered behavior into the emulator, it is because we have examined the corresponding call site in libc, and we were able to conclude that the lie completely satisfies that code path. This approach occasionally backfires when a different call site finds the lie unconvincing, but these occurrences are rare.

System configuration tools are unlikely to port well, since our approach destroys tight application coupling, for example by neutering `dbus`. We accept this limitation as fundamental to Embassies's goal of making apps more autonomous.

Embassies presently has only paper designs for audio and GPU facilities. Apps that integrate multiple programs with `fork()` are not currently supported (§3.9).

6.3 Inter-Application Protocols

This paper focuses on moving apps from a rich, trusting, shared environment to the isolated picoprocess. However, interesting apps still communicate with the outside world. Some inter-app communication is already based on IP: The apps we used discover printers and send jobs with the Internet Printing Protocol [12], so printing works correctly without special support.

However, how should apps replace communication patterns once done locally? For example, suppose one app produces data another app wishes to read. We expect such communications, once supplied by a complex trusted platform (e.g., the OS), to be replaced by IP-based protocols. Just as in the Internet, IP-based protocols are bilateral: Both participants have the opportunity to decide how much of the protocol they are willing to implement, and to select vulnerability-resistant implementations. The Embassies paper [16] addresses this question in greater detail.

7 Evaluation

7.1 Porting Effort

The most salient proof of effectiveness for our techniques is in the results: We are able to run many rich apps *without even recompiling* them (Figure 3). Instead, we binary-rewrite glibc to redirect the POSIX interface, use libraries as unmodified binaries, and relink the top-most executable to make it relocatable. That such non-invasive techniques are successful with eight interactive apps built on disparate library stacks is strong evidence that they will generalize easily to most interactive apps.

Figure 4 shows lines of code [30] in the components and patches to existing programs. Most of the effort is in the VFS implementation in the emulator.

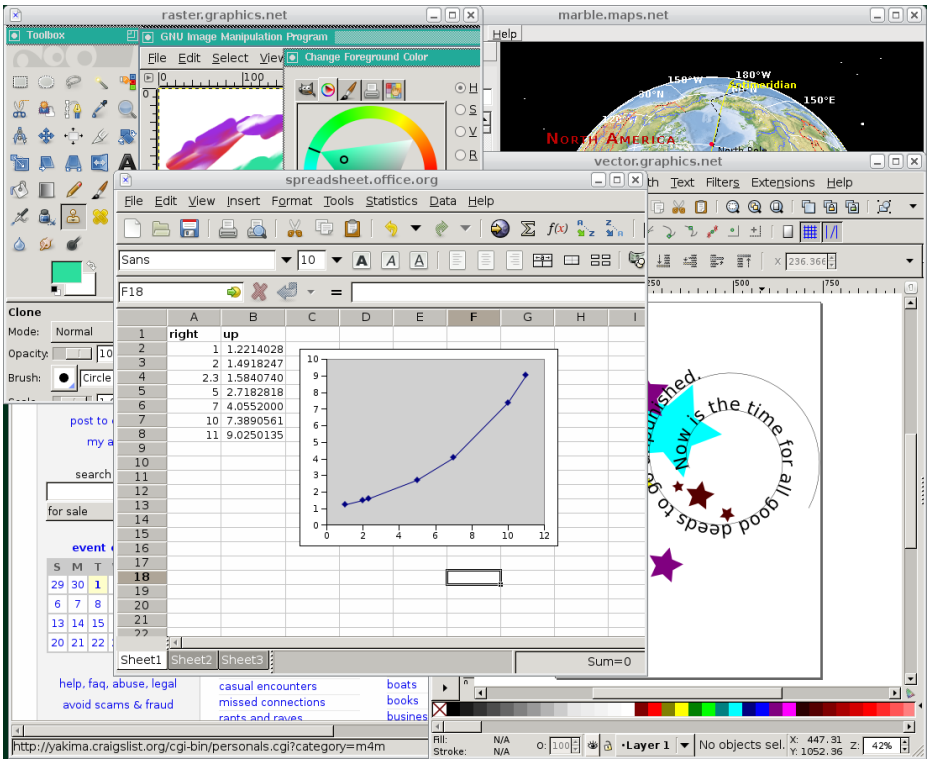

Figure 3: **POSIX emulation handles diverse, rich applications,** *e.g., the Midori Web renderer, Gimp, Marble, Inkscape, and Gnumeric. Not shown are Abiword, Gnucash, and Hyperoid.*

component	SLOC
emulator	29156
zone host	1328
lwIP patches	477
X patches	660
twm patches	0

Figure 4: **Lines of code** in system components.

7.2 Performance

For compute-bound tasks, the emulator is not involved, and apps run at native speeds. We verified this by running, on both Linux and Embassies, image rotations in Gimp and a subset of the SunSpider JavaScript benchmark [29] in Midori. As anticipated, in both cases the difference is negligible, within 2% ($c_v = 1\%$).

Informally, we have observed some emulated activities run faster than their Linux equivalents. For example, filesystem interactions with temporary files outperform Linux because they avoid a kernel-mode transition.

The application launch mechanism precludes the use of the OS buffer cache, but we recapture much of that performance in the Embassies environment [14, Fig. 14]. App starts are 50–100 ms slower than Linux; the largest bottleneck is verifying the integrity of fetched content.

7.3 Coverage

We have demonstrated a layer that emulates a small subset of Posix behavior, and we have shown this to be sufficient to run a diverse set of productivity apps. However, perhaps exercising the apps more aggressively or running additional productivity apps would require substantially more Posix-level emulation. To bound the degree to which more emulation could be required, we compare the set of syscalls visited dynamically with the set reachable statically. This analysis is approximate, because some syscalls (e.g., `ioctl`) aggregate multiple behaviors, and our static analysis tool is rather coarse.

Figure 5 shows the results. Columns are syscall numbers (sorted for contiguity); rows are applications. The upper eight apps are those we support (Table 1); the lower eight are other Linux apps to aid extrapolation. System calls in region (d) are supported as described in this paper: as meaningful, failure-oblivious, neutered, or deluded calls. Syscalls in region (x) are observed dynamically when the app is run on Linux but not when run on our emulator. For example, because `shmget` is neutered in the emulator, `shmat` and `shmctl` never appear dynamically. If the lower eight apps in Figure 5 were run on our emulator, these syscalls might be obviated for analogous reasons, but we do not know this for certain.

The 27 syscalls in region (s) are reachable statically, but not observed dynamically. Five are never called because a better version is emulated (`stat64` for `stat`). Another 7 are trivial variations of existing emulation (`ppoll` for `poll`), and 12 (those unique to muse and stella) are neuterable (`mlock`, `setgid32`). We saw 3 calls in the lower eight apps that likely require implementation: `utimes`, `symlink`, and `mknod`.

Our static analysis is imperfect, as evidenced by ten calls reached dynamically but not discovered statically.

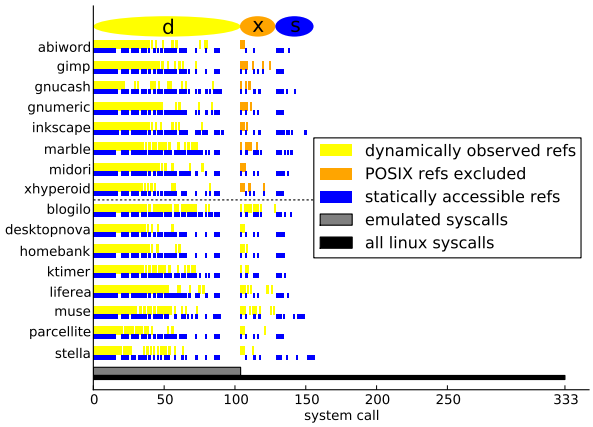

Figure 5: **An approximate static coverage check validates** *that emulating most Linux system calls is not necessary.*

Nonetheless, the broad agreement between static analysis and dynamic observation suggests that our emulation is largely complete. Of Linux's 333 syscalls, 203 are observed neither by dynamic tracing nor by static analysis. This lends credence to our claim that a broad range of productivity apps can be supported without emulating the majority of Linux functionality.

8 Related Work

8.1 Application Models

Java was offered as an alternative to the clunky mid-1990s web programming interface [10]. Absent native code, Java had to either rewrite every framework an app could want, or import and abstract existing frameworks as native libraries. Practicality demanded applying the latter technique; even the early UI toolkit AWT [33] abstracted over the host UI at a high level. The result was a Java client with a complex implementation that shared the host's vulnerabilities, and isolation that depended on a complex and growing security interface [21].

As Java largely failed to replace the HTML web app model, HTML thrived, evolving a notion of isolation [15, 32] fundamental to web apps. However, pressure to enhance functionality has progressively grown client complexity, undermining the promise of isolation [16].

The Slinky system proposed distributing POSIX apps as static binaries, enabling app developers to precisely specify their dependencies [4]. They extended the Linux kernel to detect and exploit implicit page sharing while preserving the semantics of static executables. Their approach treats shared libraries as a configuration problem. It inspired our work; we extend the Slinky insight to autonomy-preserving isolation against adversarial neighboring apps. This not only requires avoiding late-bound library sharing, but also demands eliminating the complex shared graphics stack (X or an HTML DOM renderer). Since simplicity is a priority, we eliminate even the shared buffer cache, requiring a sharing implementation different than that used in Slinky [14].

8.2 Porting Applications

Several years ago, our Xax project [13] demonstrated that rich stacks of libraries could be readily transplanted from a conventional operating system environment to provide useful functionality even from inside a picoprocess attached to a web browser. This paper reports on a more thorough implementation that supports complete, rich, interactive applications. Xax gave a high-level overview of the porting effort, enumerating five categories of techniques used to emulate the missing OS or to trigger alternative behavior in the transplanted library. This paper aims to completely demystify the process.

The Drawbridge effort demonstrated that similar techniques could be used for code based on the Windows commodity OS stack [22]; that project required introducing additional techniques, such as hoisting the GDI graphics rasterizing library from the OS kernel to become a library inside the picoprocess. The Drawbridge system assumes a non-minimal host that includes a file system, buffer cache, and TCP stack.

The task at hand is reminiscent of the Exokernel's motto, "exterminate all operating system abstractions" [18]. Like Exokernel, Embassies minimizes abstractions in the host platform; but where the Exokernel evicted abstractions to expose new performance opportunities, Embassies aims to produce a simple, rarely-changing host with a minimal attack surface. Therefore, Exokernel techniques, such as those for sharing storage, do not translate well to Embassies apps.

Google's Native Client system [31] includes ports of dozens of libraries, but does not support complete interactive applications. The difference in target assumption–that applications will run as web plug-ins, rather than replacing web apps altogether–has led the project to a different ABI, security model, and execution model. These choices necessitate a modified C compiler, which in turn requires fussing with libraries' build environment (§6.1), a task we found difficult to scale. However, once those issues are resolved, the approach in the present paper should readily enable the conversion of POSIX apps into NaCl plug-ins.

9 Conclusion

This paper showed how to support rich POSIX applications on top of a minimal picoprocess interface. Such support can be achieved by providing a POSIX emulation layer and by binding existing programs, like lwIP, `X`, and `twm` into the application itself. The POSIX emulation layer is not nearly as complicated as a conven-

tional POSIX implementation (e.g., Linux); in fact, this paper exhaustively lists every syscall emulated and every program adaptation required. Such emulation is possible in part because many POSIX functions exist to support scalability and performance more relevant to server applications (e.g., databases and web servers) and hence are unused by interactive apps. Thus, not only is it feasible to adapt POSIX applications to a sparse environment, it is reproducible. We hope these results will encourage others to adapt the existing world of rich POSIX-based applications to even the most minimal of client execution environments.

References

[1] ANDROID OS. http://www.android.com/.

[2] APPLE. iOS6, 2013. http://www.apple.com/iphone/.

[3] BARTH, A., JACKSON, C., REIS, C., AND THE GOOGLE CHROME TEAM. The security architecture of the Chromium browser. http://www.adambarth.com/papers/2008/barth-jackson-reis.pdf, 2008.

[4] COLLBERG, C., HARTMAN, J. H., BABU, S., AND UDUPA, S. K. Slinky: static linking reloaded. In *USENIX ATC* (2005).

[5] COX, R. S., GRIBBLE, S. D., LEVY, H. M., AND HANSEN, J. G. A safety-oriented platform for Web applications. In *IEEE Symp. on Security & Privacy* (2006).

[6] DREPPER, U. Futexes are tricky. Tech. rep., Red Hat, Nov. 2011.

[7] DUNKELS, A. lwIP - a lightweight TCP/IP stack. http://savannah.nongnu.org/projects/lwip/, 2013.

[8] ECMA. Standard ECMA-262: ECMAScript language specification. http://www.ecma-international.org/publications/standards/Ecma-262.htm, June 2011.

[9] FESKE, N., AND HELMUTH, C. A Nitpicker's guide to a minimal-complexity secure GUI. In *IEEE ACSAC* (2005).

[10] GOSLING, J., JOY, B., AND STEELE, G. *Java™ Language Specification*. Addison-Wesley, 1996.

[11] GRIER, C., TANG, S., AND KING, S. T. Secure web browsing with the OP web browser. In *IEEE Symposium on Security and Privacy* (2008).

[12] HASTINGS, T., HERRIOT, R., DEBRY, R., ISAACSON, S., AND POWELL, P. Internet Printing Protocol/1.1: Model and Semantics. RFC 2911 (Proposed Standard), Sept. 2000. Updated by RFCs 3380, 3382, 3996, 3995.

[13] HOWELL, J., DOUCEUR, J. R., ELSON, J., AND LORCH, J. R. Leveraging legacy code to deploy desktop applications on the web. In *OSDI* (2008).

[14] HOWELL, J., ELSON, J., PARNO, B., AND DOUCEUR, J. R. Missive: Fast appliance launch from an untrusted buffer cache. Tech. Rep. MSR-TR-2013-9, Microsoft Research, Jan. 2013.

[15] HOWELL, J., JACKSON, C., WANG, H. J., AND FAN, X. MashupOS: Operating system abstractions for client mashups. In *HotOS* (May 2007).

[16] HOWELL, J., PARNO, B., AND DOUCEUR, J. Embassies: Radically refactoring the web. In *NSDI* (2013).

[17] JANG, D., VENKATARAMAN, A., SAWKA, G. M., AND SHACHAM, H. Analyzing the crossdomain policies of Flash applications. In *IEEE Web 2.0 Security and Privacy Workshop (W2SP)* (2011).

[18] KAASHOEK, M. F., ENGLER, D. R., GANGER, G. R., NO, H. M. B., HUNT, R., MAZIÈRES, D., PINCKNEY, T., GRIMM, R., JANNOTTI, J., AND MACKENZIE, K. Application performance and flexibility on Exokernel systems. In *SOSP* (1997).

[19] MICKENS, J., AND DHAWAN, M. Atlantis: Robust, extensible execution environments for Web applications. In *SOSP* (2011).

[20] MICROSOFT. Silverlight. http://www.microsoft.com/silverlight/.

[21] NEVILLE, P. S. Mastering Java security policies and permissions. http://www2.sys-con.com/itsg/virtualcd/java/archives/0501/neville/index.html, 2004.

[22] PORTER, D. E., BOYD-WICKIZER, S., HOWELL, J., OLINSKY, R., AND HUNT, G. C. Rethinking the library OS from the top down. In *ASPLOS* (2011).

[23] REIS, C., AND GRIBBLE, S. D. Isolating Web Programs in Modern Browser Architectures. In *ACM EuroSys* (2009).

[24] RINARD, M., CADAR, C., DUMITRAN, D., ROY, D. M., LEU, T., , AND BEEBEE, JR., W. S. Enhancing server availability and security through failure-oblivious computing. In *OSDI* (2004).

[25] TANG, S., MAI, H., AND KING, S. T. Trust and Protection in the Illinois Browser Operating System. In *OSDI* (2010).

[26] WANG, H. J., FAN, X., JACKSON, C., AND HOWELL, J. Protection and communication abstractions for web browsers in MashupOS. In *SOSP* (Oct. 2007).

[27] WANG, H. J., GRIER, C., MOSHCHUK, A., KING, S. T., CHOUDHURY, P., AND VENTER, H. The multi-principal OS construction of the Gazelle web browser. In *USENIX Security Symposium* (2009).

[28] WANG, H. J., MOSHCHUK, A., AND BUSH, A. Convergence of desktop and web applications on a multi-service OS. In *USENIX HotSec Workshop* (2009).

[29] WEBKIT. SunSpider JavaScript Benchmark. Version 0.9.1 at http://www.webkit.org/perf/sunspider/sunspider.html, 2012.

[30] WHEELER, D. A. SLOCCount. Software distribution. http://www.dwheeler.com/sloccount/.

[31] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security & Privacy* (2009).

[32] ZALEWSKI, M. Browser security handbook: Same-origin policy. Online handbook. http://code.google.com/p/browsersec/wiki/Part2.

[33] ZUKOWKSI, J. *Java AWT Reference*. O'Reilly, 1997.