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 complexity out of the client platform and replacing the required behavior inside 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 [5, 13, 15, 17, 21, 24, 27–30, 32] and deployed systems [1–3, 25] 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 amongst, just as web developers choose amongst an ever evolving set of app frameworks on the server.

This minimal-client model must simultaneously achieve two properties: richness and security [6]. First, the model must support applications as rich 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.

Second, the model must support richness while keeping the isolation interface tiny and well-specified, to admit a correct client implementation. 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 [18]. 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 [9, 11, 12, 19, 22]. This complexity undermines any claims to provide strong security.

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, all implemented over the course of decades.

However, our experience suggests this task is far easier than you 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 source of 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 just how tractable it is to port rich POSIX apps to a minimal environment, thus enabling them to run on a multitude of minimal client hosts [15, 18, 20, 24, 32]. We give a full accounting 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, in order to eliminate any ambiguity about complexity hidden under the hood, and to enable reproducibility. 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 [18], 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 [7] synchronization scheduling primitive, the `zutex`. `zutex_wait` 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.

### Crypto 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’s high seas, apps build up integrity and privacy themselves using cryptography. To underscore these semantics, all communication in Embassies—remote or local—is actually done via IP.

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

### User Interface

- `ui_event` dequeues and returns a UI event (keystroke or pointer motion), returning 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. Viewport management is beyond scope, the details elided here.
- `map_canvas` allocates a framebuffer to back a viewport. This allocation is distinguished from `allocate_memory` to enable fast pixel blitting.

## Table 1: A variety of rich, functional apps transplanted to run in a minimal native picoprocess.

<table>
<thead>
<tr>
<th>Application</th>
<th>Function</th>
<th>Num. Libraries</th>
<th>Example Libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abiword</td>
<td>word processor</td>
<td>63</td>
<td>Pango, Freetype</td>
</tr>
<tr>
<td>Gimp</td>
<td>raster graphics</td>
<td>55</td>
<td>Gtk, Gdk</td>
</tr>
<tr>
<td>GnuCash</td>
<td>personal finances</td>
<td>101</td>
<td>Gnome, Enchant</td>
</tr>
<tr>
<td>Gnumeric</td>
<td>spreadsheet</td>
<td>54</td>
<td>Gtk, Gdk</td>
</tr>
<tr>
<td>Hyperoid</td>
<td>video game</td>
<td>6</td>
<td>svgalib</td>
</tr>
<tr>
<td>Inkscape</td>
<td>vector drawing</td>
<td>96</td>
<td>Magick, Gnome</td>
</tr>
<tr>
<td>Marble</td>
<td>3D globe</td>
<td>73</td>
<td>KDE, Qt</td>
</tr>
<tr>
<td>Midori</td>
<td>HTML/JS render</td>
<td>74</td>
<td>webkit</td>
</tr>
</tbody>
</table>

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 [18], 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 [7] synchronization scheduling primitive, the `zutex`. `zutex_wait` 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.

### Crypto 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’s high seas, apps build up integrity and privacy themselves using cryptography. To underscore these semantics, all communication in Embassies—remote or local—is actually done via IP.

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

### User Interface

- `ui_event` dequeues and returns a UI event (keystroke or pointer motion), returning 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. Viewport management is beyond scope, the details elided here.
- `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 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 composes 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.

Our POSIX-emulator loads and runs applications expecting this rich POSIX environment by cleverly repurposing existing libraries and programs on top of the minimal services the execution environment provides (§2). Figure 1 gives an overview of how emulator maps the entire POSIX interface down to the picoprocess interface that Embassies provides.

Below, we describe the emulation in detail, starting with application launch.

3.1 Application Launch

Embassies provides minimal support for app launch. The application vendor specifies a single bit string called the boot block. The host maps the boot block into an address space at an arbitrary address, provides it a minimal stack, and places in a register the address of a dispatch table for the Embassies ABI (§2).

Thus, when it starts, our POSIX emulator’s first steps are to relocate its own symbols (since it is loaded at an arbitrary offset) and to allocate an adequate stack. Then, it establishes a virtual file system (§3.3) and a dispatch function that emulates the Linux POSIX syscall interface.

Next, the emulator must load the app and its libraries into memory. Rather than implement this functionality from scratch, the emulator repurposes the standard Linux loader, ld-linux.so.

In Linux, a POSIX application is specified as an ELF binary. To launch it, the kernel interprets the ELF format to map the binary into memory, and also maps in the loader, ld-linux.so, to “interpret” the app binary. The loader gets control first, enumerates dynamic library references from the app, maps those libraries into memory, links the whole jumble together, and jumps to the app’s entry point.

Embassies, however, does not provide a file system from which to map files, much less a kernel willing to parse ELF binaries. Instead, our emulator includes a copy of ld-linux.so. Besides functioning as an interpreter, ld-linux.so can accept the app path name as an argument and map the app itself; this is how the emulator invokes it. Of course, this only succeeds because the ld-linux.so’s POSIX calls 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 (a few constants that soothe the libraries into thinking they are 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 seen it; 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 mmaped 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 replaces the function of the POSIX /tmp scratch directory, 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, from 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 [16]. The reply appears in memory as a single (jumbo) IP packet. The emulator ensures integrity by comparing the image’s hash to a constant hash value embedded in the boot block.

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 further-required 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 has sufficed 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...
Table 2: These socket calls plumb through the VFS interface to either the Unix named pipes implementation or the TCP stack.

<table>
<thead>
<tr>
<th>accept</th>
<th>recvfrom</th>
</tr>
</thead>
<tbody>
<tr>
<td>bind</td>
<td>recvmsg</td>
</tr>
<tr>
<td>connect</td>
<td>send</td>
</tr>
<tr>
<td>getpeername</td>
<td>sendmsg</td>
</tr>
<tr>
<td>getsockname</td>
<td>sendto</td>
</tr>
<tr>
<td>getsockopt</td>
<td>setsockopt</td>
</tr>
<tr>
<td>listen</td>
<td>shutdown</td>
</tr>
<tr>
<td>recv</td>
<td>socket</td>
</tr>
</tbody>
</table>

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 `mmaps` (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 fill the specified region with zeroes for the binary’s `.bss` section or copy in the contents of the `mapped` 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 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 a target address of the kernel’s choosing. 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.

3.4.4 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 `mapped` 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`ed.

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 vestigial calls. Instead, as a workaround, the emulator assumes that virtual memory is free, 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. It fires a zutex, and hence 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. Making the multiplexer race-free was tricky; a careful test framework is advised.

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 [7]. 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 of the 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 fall back 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 consumes IP to fetch its image (§3.3) and for querying time servers (§3.5). The emulator’s network stack demultiplexes IP and UDP.

3.8 Threads

POSIX uses clone to express both thread creation and process fork (§3.9). The emulator pattern-matches the thread-create 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 of the current thread’s stack. This is exposed in Linux POSIX through pseudofiles in /proc.

For example, Libwebkit’s JavaScriptCore garbage collector queries libpthread for the stack base (the bottom of the stack), and adds RLIMIT_STACK to it. libpthread, in turn, queries /proc/self/maps to learn the stack top from the kernel, and subtracts RLIMIT_STACK from that to produce the stack bottom. So, as it turns out, the value of RLIMIT_STACK is immaterial; it cancels out. But libthreads still sanity-checks the call to getrlimit, so some plausible value must be supplied. 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. To know where each thread’s stack resides, the emulator records each stack as its thread is created by the clone syscall. This works for all but the main thread; fortunately, that stack is known because the emulator allocated it (§3.1).

3.9 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’ 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 possibility, 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. Then exec call launches a new zone, or if fault-containment is desired, a new pico-process.

Alternatively, the idiom is usually implemented in a widely-used library, such as glib’s gspawn, where the semantics are well-defined and map cleanly onto a create-zone or create-pcprocess operation.
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 [26].

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 satiate 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 “/”.

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 memory allocations 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.

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, or the dbus desktop bus. We satisfy such apps by including these services inside the apps that need them, rather than add them to the client kernel’s TCB (which would make them a part of every app’s TCB).

X, twm, and dbus are designed as independent POSIX processes. Rather than modify them into libraries, the expedient path was to create a general mechanism for loading multiple programs into a single picoprocess. This is easier than it sounds, since the programs allocate memory and emulated file descriptors separately, carving the resource namespaces into interleaving partitions. We call such partitions “zones” (Figure 2).
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.

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 synchronously. 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

Because the goal of Embassies is to minimize the client kernel’s TCB, the complex security boundary of X is intolerable. While we do add an X zone in apps that require a display, we do not make use of its security-sensitive multiplexing functions. Instead, X is used to rasterize the app’s display, and the rendered frame buffer is blitted to the user’s display through Embassies’s pixel-level UI interface.

Some apps, like Gimp, use a plethora of palettes. 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 [10].

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 satiate 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. To preserve our sanity, we push some 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 user interfaces 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 [8] 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_mountvfs.

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 the most valuable debugging strategies we used.
Obviously, 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 such channels.

Since almost all 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. That means we can connect `gdb` to it.

However, `gdb` is not immediately useful, because it 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 have 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 triggers the generation of 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 readonly 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 serve as a manifest for generating the app image.

6 Discussion

6.1 Dynamic vs. Static Linking

We could have avoided the need for `ld-linux.so` by statically linking the applications; certainly that fits well with the app distribution model. However, that approach would require relinking every app to all of its libraries statically. Our experience with Xax [15] was that modifying a package’s build system was often a nightmare, generally much harder than modifying the source code and using the package’s build system to remake it.

The expedient of using dynamic linking means that we can use almost every library in binary form; we need not even recompile the source.

6.2 Inter-Application Protocols

This paper focuses on the challenge of moving apps from a rich, trusting shared environment to the isolated picoprocess. However, interesting apps still communicate with the outside world. Some inter-application communication is already based on IP: the apps we used discover printers and send jobs with the Internet Printing Protocol [14]; printing just worked correctly without any special support.

However, how should apps replace communication patterns once done locally? For example, if every web site brings its own HTML renderer, how does the back button work, and how does one app draw in the IFRAME of its parent? In large strokes, we expect such communications, once supplied by a complex trusted platform (e.g. the browser), 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 [18] addresses this question in much greater detail.

7 Evaluation

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 topmost executable to make it relocatable. That such non-invasive techniques are successful with eight interactive apps built on divergent library stacks is strong evidence that the technique will generalize easily to most interactive apps.
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 requires not only 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 an implementation to sharing different than that used in Slinky [16].
8.2 Porting Applications

Several years ago, our Xax project [15] demonstrated that rich stacks of libraries could be readily transplanted from a conventional operating system environment to provide useful functionality even from inside a pico-process 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 trigger alternative behavior in the transplanted library. Despite the overview, colleagues often described the effort as a one-off “miracle”. This paper aims to demystify the process.

The Drawbridge effort demonstrated that similar techniques could be used for code based on the Windows commodity OS stack [24]; 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” [20]. Like Exokernel, Embassies minimizes abstractions in the host platform; but where the paper evicted abstractions to expose new performance opportunities, Embassies aims to produce a simple, rarely-changing host with minimal vulnerability surface. Therefore, Exokernel techniques such as those for sharing storage do not translate well to Embassies apps.

The Native Client team has ported dozens of libraries, but have not moved up the stack to 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 NaCl team to a different ABI, security model, and execution model. These choices led to the requirement to require a modified C compiler, which in turn requires fusing with libraries’ build environment (§6.1), a task we found difficult to scale. However, once those issues are resolved, the approach presented here should readily enable the adaptations of POSIX apps to 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 conventional 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 anticipate these results will encourage others to adapt the word of rich existing POSIX-based applications to even the most minimal of client execution environments.

References


