Leveraging Legacy Code to Deploy Desktop Applications on the Web

John R. Douceur, Jeremy Elson, Jon Howell, and Jacob R. Lorch
Microsoft Research

Abstract

Xax is a browser plugin model that enables developers to leverage existing tools, libraries, and entire programs to deliver feature-rich applications on the web. Xax employs a novel combination of mechanisms that collectively provide security, OS-independence, performance, and support for legacy code. These mechanisms include memory-isolated native code execution behind a narrow syscall interface, an abstraction layer that provides a consistent binary interface across operating systems, system services via hooks to existing browser mechanisms, and lightweight modifications to existing tool chains and code bases. We demonstrate a variety of applications and libraries from existing code bases, in several languages, produced with various tool chains, running in multiple browsers on multiple operating systems. With roughly two person-weeks of effort, we ported 3.3 million lines of code to Xax, including a PDF viewer, a Python interpreter, a speech synthesizer, and an OpenGL pipeline.

1 Introduction

Web applications are undergoing a rapid evolution in functionality. Whereas they were once merely simple dynamic enhancements to otherwise-static web pages, modern web apps[1] are driving toward the power of fully functional desktop applications such as email clients (Gmail, Hotmail, Outlook Web Access) and productivity apps (Google Docs). Web applications offer two significant advantages over desktop apps: security—in that the user's system is protected from buggy or malicious applications—and OS-independence. Both of these properties are normally provided by a virtual execution environment that implements a type-safe language, such as JavaScript, Flash, or Silverlight. However, this mechanism inherently prohibits the use of non-type-safe legacy code. Since the vast majority of extant desktop applications and libraries are not written in a type-safe language, this enormous code base is currently unavailable to the developers of web applications.

Our vision is to deliver feature-rich, desktop-class applications on the web. We believe that the fastest and easiest way to create such apps is to leverage the existing code bases of desktop applications and libraries, thereby exploiting the years of design, development, and debugging effort that have gone into them. This existing code commonly expects to run in an OS process and to access OS services. However, actually running the code in an OS process would defeat the OS-independence required by web apps; and it would also impede code security, because large and complex OS system-call (or *syscall*) interfaces are difficult to secure against privilege-escalation vulnerabilities [15].

There is thus a trade-off between **OS-independence**, **security**, and **legacy support**. No existing web-app mechanism can provide all three of these properties. Herein, we show that is possible to achieve all three, and further to achieve native-code **performance**.

In particular, we propose eliminating the process's access to the operating system, and instead providing only a very narrow syscall interface. A sufficiently narrow interface is easy to implement identically on different operating systems; and it is far easier to secure against malicious code. However, it may not be obvious that a process can do much useful work without an operating system to call, and it is even less clear that legacy code could easily be made to work without an OS.

Surprisingly, we found that with minimal modification, legacy libraries and applications with large code bases and rich functionality can indeed be compiled to run on a very simple syscall interface. We demonstrate this point by running the GhostScript PDF viewer, the eSpeak speech synthesizer, and an OpenGL demo that renders 3D animation. In total, it took roughly two person-weeks of effort to port 3.3 million lines of code to use this simple interface. This existing code was written in several languages and produced with various tool chains, and it runs in multiple browsers on multiple operating systems.

We achieved these results with *Xax*, a browser plugin model that supports legacy code in a secure and OS-independent manner, and which further provides native-code performance as required by feature-rich applications. Xax achieves these properties with four mechanisms:

- The *picoprocess*, a native-code execution abstraction that is secured via hardware memory isolation and a very narrow system-call interface, akin to a streamlined hardware virtual machine
- The Platform Abstraction Layer (PAL), which provides an OS-independent Application Binary Interface (ABI) to Xax picoprocesses
- Hooks to existing browser mechanisms to provide applications with system services—such as network communication, user interface, and local storage—that respect browser security policies
- Lightweight modifications to existing tool chains and code bases, for retargeting legacy code to the Xax picoprocess environment

The key principle behind Xax is that the browser already contains sufficient functionality to support the necessary system services for running legacy code. Xax provides this support with its novel combination of four mechanisms and its specific design decisions within each mechanism. Together, these choices achieve our goal of high-performance support for legacy desktop code in secure, OS-independent web applications.

Xax provides key pieces of a comprehensive solution to enable skilled developers to deploy actual desktop applications on the web. Although we have not yet built a large, full-featured application, we have built several moderate-sized applications using a dozen libraries and application components we have ported.

In addition, by leveraging not only existing application code and libraries but also existing development tool chains, Xax allows even moderately skilled developers to combine the conventional DOM-manipulation model of web applications with the power of existing non-web-specific code libraries, arbitrary programming languages, and familiar development tools. We demonstrate this by porting a Python interpreter to Xax and providing language bindings to JavaScript DOM functions, after which we created a social-network visualization app using unmodified Python wrappers for the graphviz library.

Finally, we show that the Xax plugin model can actually subsume other browser plugins. We demonstrate this with a basic port of the Kaffe Java Virtual Machine (JVM) into Xax. Because Kaffe runs within a Xax picoprocess, it does not add to the browser's trusted code base, unlike the standard JVM browser plugin.

The next section details the goals of Xax and contrasts with alternative approaches. Section 3 describes the four mechanisms Xax uses to achieve its goals: picoprocesses, the Platform Abstraction Layer, services via browser mechanisms, and lightweight code modification. Section 4 describes our implementations of Xax in Linux and Windows, as well as our proxy-based browser integration. Section 5 describes some of our example applications. Section 6 evaluates the four benefits of Xax described in Section 2. Sections 7 and 8 describe related and future work. Section 9 summarizes and concludes.

2 Goals and Alternatives

In this section, we detail the goals that must be satisfied to deliver desktop applications on the web, and we consider alternative mechanisms for achieving these goals.

2.1 Xax Design Goals

As previewed in the Introduction, Xax has four design goals: security, OS-independence, performance, and legacy support. For the first three, our intent is to match the benefits of existing web-app mechanisms, such as JavaScript and Flash. Xax's main benefit beyond existing mechanisms is support for legacy code.

security — The particular form of security required for web applications is protecting the client against malicious code. (For the scope of this paper, we ignore other threats such as cross-site scripting and phishing.) Part of what makes web applications attractive is that they are supposed to run safely without requiring explicit trust assumptions from the user. This stands in contrast to installed desktop applications, which have nearly unfettered access to the client machine, so users make trust assumptions whenever they install a desktop program from a CD or via the Internet. Web applications are considered safe because they execute within a sandbox that sharply restricts the reach of the program.

OS-independence — Unlike desktop apps, web applications are not tied to a particular operating system, because they do not make direct use of OS services. Instead, web apps invoke services provided by the browser or by a browser plugin, which is responsible for exporting the same interface and semantics across OS implementations. Ideally, web apps are also independent of the particular browser in which they run; however, some aspects of HTML and the JavaScript environment are not implemented consistently among browsers [22], which somewhat limits this benefit in practice. In addition, a particular web app might rely on features of a particular plugin version, so running the web app might require downloading and installing a new version of the plugin (which entails making a trust assumption).

performance — Simple web apps, such as web pages with dynamic menus, may not require much performance from their execution environments. However, the performance demands may be significant for feature-rich applications that provide functionality comparable to desktop applications, such as animated 3D rendering.

legacy support — Developing complex, feature-rich applications requires an enormous effort. A nearly essential practice for mitigating this effort is software reuse, which has been a staple of the computer industry since the idea was first proposed in 1969 [27]. Despite the fact that the past decade has seen an increasing amount of new code written in type-safe languages, the vast majority of extant software is not type-safe. Of 311 million lines of code in the SourceForge [40] repository, half are C (29%) and C++ (18%); by contrast, Java, C# and JavaScript combined account for only 17%. Large fractions of the 86 million lines [21] of Mac OS X, 200 million lines [1] of Windows Vista, and 283 million lines [36] of the Debian GNU/Linux distribution are general libraries that provide significant functionality to desktop applications; much of this legacy code could benefit the development of rich web applications.

2.2 Alternative Mechanisms

Xax achieves all four of the above goals. Many existing mechanisms for developing web apps already exist, but each falls short of these goals in at least some respects. There are also other yet-undeployed approaches one could explore; we argue that Xax has advantages over each of these other approaches.

2.2.1 Existing web-app mechanisms

There are a number of existing mechanisms for implementing web applications. This set of mechanisms cannot be totally ordered with respect to our four goals; however, we make an attempt to present them in roughly increasing order of goal satisfaction.

JavaScript is an interpreted scripting language with dynamic typing and very late binding. It is included in all major web browsers, so it has the exclusive benefit of not requiring a plugin. It provides language-based security and OS-independence. Because it is interpreted, it does not have very good performance, and the late-binding dynamic semantics of the language make it difficult to JIT and therefore slow. Conversion tools provide limited legacy support for other type-safe languages, including Python, Java, C#, and Pascal, but not for any non-type-safe language.

ActiveX controls are means for packaging client-side code that can be invoked from a web page. They execute natively, so they provide high performance. They have some legacy support for non-type-safe code, particularly for C++ code that is compiled with the Microsoft Foundation Class (MFC) library or the Active Template Library (ATL), as well as support for other languages such as Delphi and Visual Basic. ActiveX controls work only on Windows, and because they have unrestricted OS access, they provide no security against malicious code.

Type-safe intermediate-language systems include Flash, the Java Virtual Machine (JVM), and Silverlight. These all provide security via translating a type-safe source language into a type-safe intermediate language—such as bytecode—that is downloaded to the browser. The definition of the intermediate language is OS-independent, and interpreters or JIT compilers exist for all major browsers. Performance is good because of JIT compiling. Collectively, these systems support a sizeable count of type-safe languages: Flash bytecode can be generated from ActionScript and LZX. Java bytecode can be generated from Java, Python, Ruby, JavaScript, and Common Lisp. Common Language Runtime code, Silverlight 2's intermediate language, can be generated from C#, Visual Basic, Managed C++, and a number of uncommon languages. However, none of these systems can support legacy code written in a non-type-safe language, which — as observed above — is the vast majority of extant code.

2.2.2 OS processes

Since the lion's share of legacy code was written to run in an OS process and to access OS services, a natural way to support this code within a web application is to actually run it inside an OS process. This approach provides the performance of native-code execution as well as direct legacy support. However, it leads to two problems that could potentially be solved at some cost.

First, and most obviously, OS processes are not OS-independent. However, it is possible to write compatibility layers [47] that that map foreign OS calls to native OS calls. Such compatibility layers are notoriously hard to write, because OS processes require bug-for-bug binary-compatible emulation of the OS interface.

Second, OS processes provide insufficient safety for web apps, since the interface to the OS is powerful enough for the process to harm the client machine. However, it is possible to write confinement layers [16, 17, 34] that restrict the allowable system calls made by a process. Such confinement layers are also quite challenging to create, not because the mechanism is particularly difficult, but because of the subtleties in defining appropriate policies that are sufficiently liberal to permit application functionality while sufficiently restrictive to prevent security breaches [15].

More broadly, we believe that trying to pare away dangerous entry points and combinations of calling parameters from a wide and complex interface is fraught with error. As described below, Xax takes the opposite tack by starting with no interface and then adding the minimum necessary to provide useful functionality.

2.2.3 Hardware virtual machines

Another alternative is to run a legacy application, along with the OS for which it was written, inside a hardware virtual machine (VM). Modern VM technology provides impressive performance that rivals native execution speed. It achieves strong security against malicious code through a combination of isolation via hardware and communication via a virtual network interface. A VM that runs on top of multiple host OSes can provide OS-independence, and a VM that executes multiple guest OSes can provide full legacy support. Tahoma contains web browsers using VMs [8]. However, use of a VM for executing web applications leads to three concerns.

First, the virtual machine monitor (VMM), which provides the hardware emulation environment for a VM, is part of the trusted code base. Because VMMs are large and complex, they contain significant potential for security vulnerabilities [32].

Second, virtual machine images are very large, because they include not only the application and libraries but also a full OS. For instance, we measured that a sparsely configured Debian system fetches over 25 MB of pages merely to boot. Given typical wide-area connection speeds, VM images can take hours to download, even with optimizations [23, 5, 39]. It is plausible that sophisticated caching and prefetching strategies could mitigate some of this download time. In addition, the techniques we use to port apps to Xax (§3.4) could similarly be applied to reducing VM image sizes.

Third, VM technology is challenging to implement, which makes it difficult to extend to other platforms, such as mobile devices. A VMM must perform accurate emulation of kernel-mode execution, a full MMU, addressable and programmable devices, and the convoluted addressing modes employed during OS boot. This complexity is so challenging that only one fully virtualizing VM product is currently able to run multiple guest OSes and to run on multiple host OSes [45].

The complexity of VM technology can be reduced by paravirtualization [46], which entails making small changes to the guest code to reduce the emulation burden on the VM system. Such changes may obviate some of the more cumbersome addressing modes or eliminate the need for binary rewriting to mask unvirtualizable machine features.

We observe that the paravirtualization concept can be taken to an extreme. Rather than merely modifying the guest OS, one can eliminate the guest OS along with some of the guest libraries, and then make changes to the guest application that enable it to run on a dramatically less-functional substrate. Such an approach substantially reduces the size and complexity of the VMM, since it need not emulate physical devices, the MMU, or CPU kernel mode. In addition, this approach dramatically reduces the size of VM images, since they contain little more than application code. Moreover, reducing the size of the VMM reduces the size of the trusted code base, thereby improving security. Such extreme paravirtualization is one way to view the Xax picoprocess architecture, which we describe next.

3 Mechanisms

Section 2.1 itemized four design goals, and the present section describes the four mechanisms by which Xax achieves these goals. Despite the agreement in number, there is not a one-to-one correspondence between the goals and the mechanisms. Security is provided by picoprocesses and browser-based services; OS-independence is provided by picoprocesses and the Platform Abstraction Layer; performance is provided by picoprocesses; and legacy support is provided by lightweight code modification.

3.1 Picoprocesses

The core abstraction in Xax is the *picoprocess*. As described in the previous section, a picoprocess can be thought of as a stripped-down virtual machine without emulated physical devices, MMU, or CPU kernel mode. Alternatively, a picoprocess can be thought of as a highly restricted OS process that is prevented from making kernel calls. In either view, a picoprocess is a single hardware-memory-isolated address space with strictly user-mode CPU execution and a very narrow interface to the world outside the picoprocess, as illustrated in Figure 1.

Picoprocesses are created and mediated by a browser plugin called the *Xax Monitor*. Like a virtual machine monitor (in the VM analogy) or an OS kernel (in the OS process analogy), the Xax Monitor is part of the browser's trusted code base, so it is important to keep it small. The picoprocess communicates by making *xaxcalls* (analogous to syscalls) to the Xax Monitor.

Because the Xax Monitor uses OS services to create and manage picoprocesses, it is necessarily OS-specific. Moreover, to ease the implementation burden and help keep the Xax Monitor simple, we do not enforce a standard xaxcall interface. The specific set of xaxcalls, as well as the xaxcall invocation mechanism, may vary depending on the underlying OS platform. We describe some differences below in sections on our Linux (§4.2) and Windows (§4.3) implementations. In terms of functionality, xaxcalls provide means for memory allocation and deallocation, raw communication with the browser, raw communication with the origin server, access to URL query parameters, and picoprocess exit.

The simplicity of the xaxcall interface makes it very easy to implement on commodity operating systems, which assists OS-independence. This simplicity also

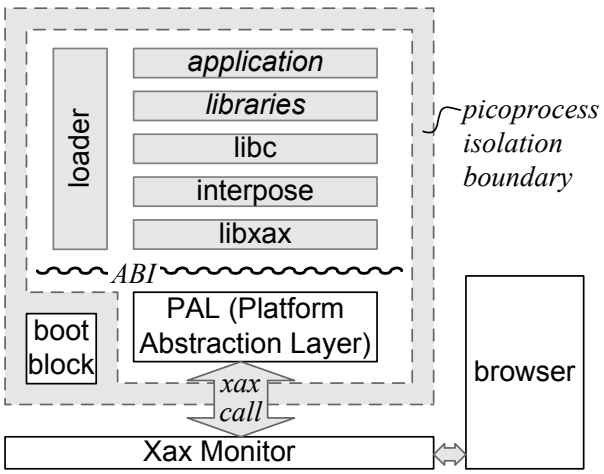

Figure 1: The Xax architecture. Everything inside the picoprocess (§3.1) isolation boundary is untrusted. The Xax Monitor mediates access to the outside world, employing existing browser mechanisms (§3.3) to implement xaxcalls from the picoprocess. The PAL (§3.2) provides a consistent Application Binary Interface (ABI) across OS platforms. Above the ABI, the specific structure can vary; the depicted structure is one we have found useful when porting code (§3.4).

aids security, since it is much easier to reason about the security aspects of a narrow interface with simple semantics than a wide interface with complex semantics. Because a picoprocess executes native code, it provides good performance. However, it is not necessarily clear that this architecture supports legacy code that was written with the expectation of running in an OS process with access to rich OS services; we address this point in §3.4 below.

3.2 Platform Abstraction Layer

As mentioned in the previous section, the xaxcall interface may vary slightly across OS platforms. For OS-independence, Xax defines a consistent *Application Binary Interface* (ABI) irrespective of the underlying OS. By necessity, the ABI varies across architectures, so the x86 ABI is different from the PowerPC ABI.

The ABI is exported by an OS-specific *Platform Abstraction Layer* (PAL), which translates the OS-independent ABI into the OS-specific xaxcalls of the Xax Monitor. The PAL is included with the OS-specific Xax implementation; everything above the ABI is native code delivered from an origin server. The PAL runs inside the Xax picoprocess, so its code is not trusted. Security is provided by the xaxcall interface (dashed border in Figure 1); the PAL merely provides ABI consistency across host operating systems (wiggly line in Figure 1).

All xaxcalls are nonblocking except for `poll`, which can optionally yield until I/O is ready. This provides sufficient functionality for user-level threading.

We herewith present the entire Xax ABI. For memory allocation and deallocation, the ABI includes the following two calls:

```
void *xabi_alloc(
  void *start, long len);
```

Map `len` zero-filled bytes of picoprocess memory, starting at `start` if specified. Return the address.

```
int xabi_free(void *start);
```

Free the memory region beginning at `start`, which must be an address returned from `xabi_alloc`. Return 0 for success or -1 for error.

As described in the next section, the picoprocess appears to the browser as a web server, and communication is typically over HTTP. When the browser opens a connection to the picoprocess, this connection can be received by the following call:

```
int xabi_accept();
```

Return a channel identifier, analogous to a Unix file descriptor or a Windows handle, connected to an incoming connection from the browser. Return -1 if no incoming connection is ready.

The picoprocess can also initiate connection to the server that provided the picoprocess application. To initiate a connection to the home server, the picoprocess uses the following call:

```
int xabi_open_url(
  const char *method,
  const char *url);
```

Return a channel identifier connected to the given URL, according to the specified `method`, which may be "get", "put", or "connect". Fetch and cache the URL according to the Same Origin Policy (SOP) rules for the domain that provided the Xax picoprocess.

The operations that can be performed on an open channel are `read`, `write`, `poll`, and `close`:

```
int xabi_read(
  int chnl, char *buf, int len);
int xabi_write(
  int chnl, char *buf, int len);
```

Transfer data on an open channel. Return the number of bytes transferred, 0 if the channel is not ready, or -1 if the channel is closed or failed.

```
typedef struct {
  int channel;
  short events;  /* requested */
  short revents; /* returned */
} xabi_poll_fd;
```

```
int xabi_poll(
  xabi_poll_fd *pfds, int npfds,
  bool block);
```
Indicate the ready status of a set of channels by updating `revents`. If `block` is true, do not return until at least one requested event is ready, thereby allowing the picoprocess to yield the processor. Return the number of events ready; do not return 0 if `block` is true.

```
int xabi_close(int chnl);
```
Close an open channel. Return 0 for success or -1 for error.

During picoprocess boot, the loader (§4.4) needs to know the URL from which to fetch the application image. We could have required a custom loader for each application, with the URL baked into the loader's image. Instead, we wrote a general loader that reads the application URL from the query parameters of the URL that launched the picoprocess. The following call, which in normally used only by the loader, provides access to these parameters. (Note that there is no corresponding xaxcall; the parameters are written into the PAL during picoprocess initialization.)

```
const char **xabi_args();
```
Return a pointer to a NULL-terminated list of pointers to arguments specified at instantiation.

Lastly, the ABI provides a call to exit the picoprocess when it is finished:

```
void xabi_exit();
```

Although the PAL runs inside the picoprocess, it is not part of the application. More pointedly, it is not delivered with the OS-independent application code. Instead, the appropriate OS-specific PAL remains resident on the client machine, along with the Xax Monitor and the web browser, whose implementations are also OS-specific. When a Xax application is delivered to the client, the app and the PAL are loaded into the picoprocess and linked via a simple dynamic-linking mechanism: The ABI defines a table of function pointers and the calling convention for the functions. For x86 architectures, this calling convention is `cdecl`; for the PowerPC, it is the one standard calling convention; and, for other architectures, no Xax ABI has yet been defined.

We have found it helpful to create a simple shim library called `libxax` that an application may statically link. `libxax` exports a set of symbols (`xabi_read`, `xabi_open_url`, etc.) that obey the function linkage convention of the developer's tool chain. The shim converts each of these calls to the corresponding ABI call in the PAL. This shim thus provides a standard Application Programming Interface (API) to Xax applications.

3.3 Services via browser mechanisms

A key Xax principle is that there is sufficient functionality within the browser to support the system services needed by web applications. In fact, we assert not only that it is sufficient for the Xax Monitor to employ the browser's functionality, but also that doing so improves the system's security. Because Xax reuses the existing security policy—and much of the mechanism—in the browser, Xax introduces no new security vulnerabilities, modulo implementation bugs in the Xax Monitor's (small) trusted code base.

The Xax Monitor has the job of providing the services indicated by the xaxcall interface. Some of these services are straightforward for the Xax Monitor to perform directly, such as memory allocation/deallocation, access to URL query parameters, and picoprocess exit. The Xax Monitor also provides a communication path to the browser, via which the Xax picoprocess appears as a web server. This communication path enables the Xax application to use `read` and `write` calls to serve HTTP to the browser. From the browser's perspective, these HTTP responses appear to come from the remote server that supplied the Xax app. It is clear that this approach is secure, since the Xax application is unable to do anything that the remote server could not have done by serving content directly over the Internet.

Using the picoprocess-to-browser communication path, the Xax application can employ JavaScript code in the browser to perform functions on its behalf, such as user interface operations, DOM manipulation, and access to browser cookies. In our applications, we have applied a common design pattern: The Xax app provides an HTML page to the browser, and this page contains JavaScript stubs which translate messages from the picoprocess into JavaScript function invocations.

It would be possible but awkward to use JavaScript for network communication. To pass through JavaScript, an application or library binary from a remote server would have to be uuencoded, encapsulated in JSON, transferred via HTTP, de-encapsulated, and decoded. To simplify this process, we provide the ABI call `xabi_open_url` to allow direct communication between a Xax picoprocess and its origin server. Both our Linux and Windows Xax Monitors provide corresponding xaxcalls that implement the primitives efficiently.

3.4 Lightweight code modification

One of our most surprising findings is how little effort it takes to port a legacy application, library, or tool chain to the minimalist Xax ABI. This is surprising because this legacy code was written to run atop an operating system, and it was not *a priori* obvious that we could eliminate the OS and still enable the legacy code to perform its main function. For instance, to enable development of

the app described in §5.2, we ported the graphviz library and the Python interpreter to Xax. Using `strace`, we saw that a quick test application makes 2725 syscalls (39 unique). Porting this code to Xax would seem to require an enormous emulation of OS functionality. However, using our lightweight modifications, we ported this million lines of code in just a few days.

Although the particular modifications required are application-dependent, they follow a design pattern that covers five common aspects: disabling irrelevant dependencies, restricting application interface usage, applying failure-oblivious computing techniques, internally emulating syscall functionality, and (only when necessary) providing real syscall functionality via xaxcalls.

The first step is to use compiler flags to disable dependencies on irrelevant components. Not all libraries and code components are necessary for use within the web-application framework, and removing them reduces the download size of the web app and also reduces the total amount of code that needs to be ported. For Python/graphviz, by disabling components such as pango and pthreads, we eliminated 699 syscalls (16 unique).

The second step is to restrict the interfaces that the application uses. For instance, an app might handle I/O either via named files or via `stdin`/`stdout`, and the latter may require less support from the system. Depending on the app, restricting the interface is done in various ways, such as by setting command-line arguments or environment variables. For Python/graphviz, we used an entry-point parameter to change the output method from "xlib" to "svg", which eliminated 367 syscalls (21 unique).

The third step is to identify which of the application's remaining system calls can be handled trivially. For example, we can often return error codes indicating failure, in a manner similar to failure-oblivious computing [35]. For Python/graphviz, it was sufficient to simply reject 125 syscalls (11 unique). Specifically, we obviate `getuid32`, `rt_sigaction`, `fstat64`, `rt_sigprocmask`, `ioctl`, `uname`, `gettimeofday`, `connect`, `time`, `fcntl64`, and `socket`.

The fourth step is to emulate syscall functionality within the syscall interpose layer (see Figure 1). For instance, Python/graphviz reads Python library files from a file system at runtime. We package these library files as a tar ball, and we emulate a subset of file-system calls using libtar to access the libraries. The tar ball is read-only, which is all Python/graphviz requires. For some of our other ported applications, we also provide read/write access to temporary files by creating a RAM disk in the interpose layer. Code in the interpose layer looks at the file path to determine whether to direct calls to the tar ball, to the RAM disk, or to somewhere else, such as a file downloaded from the origin server. For Python/graphviz, we use internal emulation to satisfy 1409 syscalls (14 unique), 943 of which fail obliviously.

The fifth and final step is to provide real backing functionality for the remaining system calls via the Xax ABI. For Python/graphviz, most of the remaining syscalls are for user input and display output, which we route to UI in the browser. We provide this functionality for the remaining 137 syscalls (11 unique). Specifically, we implement `setsockopt`, `listen`, `accept`, `bind`, `read`, `write`, `brk`, `close`, `mmap2`, `old_mmap`, and `munmap`.

The first three steps are application-specific, but for the final two steps, we found much of the syscall support developed for one app to be readily reusable for other apps. For example, we originally wrote the internally emulated tar-based file system to support eSpeak, and we later reused it to support Python. Similarly, the backing functionality for the `mmap` functions and networking functions (`listen`, `accept`, `bind`, ...) is used by all of our example applications.

For any given application, once the needed modifications are understood, the changes become mechanical. Thus, it is fairly straightforward for a developer to maintain both a desktop version and a Xax version of an app, using a configure flag to specify the build target. This is already a common practice for a variety of applications that compile against Linux and BSD and Win32 syscall interfaces.

4 Implementation

In this section, we describe the implementations of Xax on Linux and Windows, as well as our proxy-based browser integration.

Although they have some significant differences, our two implementations of Xax share much common structure. The main aspect in which they differ is in the kernel support for picoprocess isolation and communication, which we will discuss after first describing the common aspects.

4.1 Monitor, boot block, and PAL

The Xax Monitor is a user-mode process that creates, isolates, and manages each picoprocess (§3.1), and that provides the functionality of xaxcalls (§3.3). A picoprocess is realized as a user-level OS process, thus leveraging the hardware memory isolation that the OS already enforces on its processes. Before creating a new picoprocess, the Xax Monitor first allocates a region of shared memory, which will serve as a communication conduit between the picoprocess and the Monitor. Then, the picoprocess is created as a child process of the Xax Monitor process.

This child process begins by executing an OS-specific *boot block*, which performs three steps. First, it maps the shared memory region into the child process's address space, thereby completing the communication conduit. Second, it makes an OS-specific kernel call that permanently revokes the child process's ability to make subsequent kernel calls, thereby completing the isolation. Third, it passes execution to the OS-specific PAL, which in turn loads and passes execution to the Xax application.

Note that the boot block is part of the TCB, even though it executes inside the child process. The child process does not truly become a picoprocess until after the boot block has executed. At that point, the child process has no means to de-isolate itself, since this would require a kernel call but the picoprocess is prevented from making kernel calls.

After transferring control to the Xax application, the PAL (§3.2) has the job of implementing the Xax ABI by making appropriate xaxcalls to the Xax Monitor. To make a xaxcall, the PAL writes the xaxcall identifier and arguments into the shared memory region, then traps to the kernel. In an OS-specific manner (described below) the kernel notifies the Xax Monitor of the call. The Monitor then reads the shared memory, performs the indicated operation, writes the result to the shared memory, and returns control to the picoprocess.

Although the Xax Monitor has different implementations on different operating systems, it handles most xaxcalls in more-or-less the same way irrespective of OS. The `alloc` and `free` xaxcalls are exceptions to this rule, so their different implementations are described in the following two sections. For `accept`, the Xax Monitor maintains a queue of connection requests from the browser, and each call dequeues the next request. The `open_url` xaxcall makes an HTTP connection to a remote resource; the returned channel identifier corresponds to either a socket handle or a file handle, depending on whether the requested data is cached. The I/O calls `read`, `write`, `poll`, and `close` are implemented by reading, writing, polling, and closing OS file descriptors on sockets and files. The `exit` xaxcall simply terminates the child process.

4.2 Linux kernel support

Our Linux implementation involves no custom kernel code. Instead, it makes use of the Linux kernel's `ptrace` facility, which enables a process to observe and control the execution of another process.

As described above, the boot block makes a kernel call to revoke the child process's ability to make subsequent kernel calls. In our Linux implementation, this is done by calling `ptrace(TRACE_ME)`, which causes the kernel to intercept the entry and exit of every subsequent syscall, transferring control to the Xax Monitor parent process. On entry to a syscall, the Xax Monitor normally replaces whatever system call the child process requested with a harmless system call (specifically, `getpid`) before releasing control to the kernel. This prevents the child process from passing a syscall to the OS.

Syscalls are also legitimately used by the PAL to signal a xaxcall. Thus, when `ptrace` notifies the Xax Monitor of an entry to a syscall, the Monitor checks whether the shared memory contains a legitimate xaxcall identifier and arguments. If it does, the Xax Monitor performs the operation and returns the result, as described above. If the xaxcall is a memory-management operation (`alloc` or `free`), it has to be handled specially, because Linux does not provide a mechanism for a process to allocate memory on behalf of another process. So, in this case, the Xax Monitor does not overwrite the syscall with `getpid`. Instead, it overwrites the syscall with `mmap` and a set of appropriate arguments. Since the return from the syscall is also intercepted by `ptrace`, the Xax Monitor has an opportunity to write a return value for the `alloc` xaxcall into the shared memory, based on the return value from the `mmap` syscall.

Use of an existing kernel facility (`ptrace`) enables our Linux implementation to be deployed without kernel-module installation or root privilege. However, it entails a performance hit, because every xaxcall requires three syscalls from the Xax Monitor: one to swap out the syscall with `getpid` or `mmap`, a second to enter the kernel, and a third to resume the picoprocess. More importantly, if the Xax Monitor fails and exits without proper signal handling, the child process may continue to run without having its syscalls intercepted [34]. This failure condition could turn the picoprocess back into a regular OS process, which would violate security.

These performance and security problems could be mitigated by using a custom kernel module instead of `ptrace`. In the future, we intend to employ this approach, and we have already done so in our Windows implementation.

4.3 Windows kernel support

In our Windows implementation, when the child process's boot block makes a kernel call to establish an interposition on all subsequent syscalls, it makes this call to a custom kernel module, *XaxDrv*. Because every Windows thread has its own pointer to a table of system call handlers, XaxDrv is able to isolate a picoprocess by replacing the handler table for that process's thread. The replacement table converts every user-mode syscall into an inter-process call (IPC) to the user-space Xax Monitor. For a syscall originating from kernel mode (e.g., for paging), XaxDrv passes the call through to the original handler, preserving the dispatcher's stack frame for the callee's inspection.

When the Xax Monitor receives an IPC, it reads the xaxcall identifier and arguments from the shared memory and performs the operation. Unlike the Linux case, no special handling is required for memory-management operations, because Windows `NtMapViewOfSection` allows the Monitor to map memory on behalf of its child process.

Although XaxDrv has to be ported to each version of Windows on which it runs, the changes are minimal, involving two constant scalars and a constant array: (1) the offset in the kernel thread block for the pointer to the syscall handler table, (2) the count of system calls, and (3) for each system call, the total parameter byte count. This information is readily available from the kernel debugger in the Windows Driver Kit [28]. We have ported XaxDrv to Windows XP, Windows Vista, and Windows Server 2008.

An alternative implementation of XaxDrv could have followed the common approach [25, 38] of patching every entry in the standard system-call table. However, Microsoft discourages this practice because it transparently changes the behavior of every process in the system. Furthermore, even if the interposed handlers were to properly fall through to the original handlers, they would still add overhead to every system call.

4.4 Loaders

The Linux toolchain emits standard statically-linked Elf binaries. These Xax binaries are loaded by a small `elfLoader`. This loader reads the target binary, parses it to learn where to map its program regions, and looks up two symbols: a global symbol where the binary's copy of `libxax` expects to find a pointer to the PAL's dispatch table, and the address of the `_start` symbol. Then `elfLoader` maps the program, writes the dispatch table location into the pointer, and jumps to `_start`.

The Windows toolchain emits statically-linked `.EXE` binaries in Windows' native PE-COFF format. Our `peLoader` performs the corresponding tasks to map and launch PE executables.

4.5 Browser integration

Recall (§3.3) that the Xax application appears to the browser as part of the origin server that just happens to handle HTTP requests very quickly; this ensures that the picoprocess is governed by the Same-Origin Policy [20] just as is the origin server.

Our implementation integrates Xax into the browser via an HTTP proxy. This approach is expedient, and one implementation serves all makes of browser. The proxy passes most HTTP requests transparently to the specified host. However, if the URL's path component begins with `/_xax/`, the proxy interposes on the request to direct the request to an existing picoprocess or to create a new one. The proxy is integrated with the Xax Monitor process, and allows each picoprocess to contact its origin server via `xax_open_url`. This contact employs the same mechanism that fetches ordinary URLs, and thus obeys the SOP.

5 Examples

This section highlights several features of Xax by way of brief presentations of some application examples.

5.1 Headline Reader and 3D Demo

The only connection between a Xax picoprocess and the browser is an HTTP channel, which might seem insufficient to deliver the rich content that can be provided by other plugins. However, we present two applications that show this channel to be sufficient.

First, the Headline Reader app performs text-to-speech conversion. We ported the 25K-line eSpeak speech synthesizer to Xax and invoke it with a small wrapper app we wrote. The app produces .WAV audio clips, which are transferred to the browser via the HTTP channel and then played using the browser's standard audio helper.

Second, the 3D Demo performs real-time 3D rendering. We ported the 684K-line Mesa OpenGL library to Xax. It includes a demo which draws a 400×400-pixel 3D scene; we modified it to animate. We express the output of OpenGL as a series of PNG files, which are sequentially transferred to the browser periodically and inserted into an HTML DIV element for display. This approach is performance-limited by the time spent encoding PNG files; the Xax mesa demo renders 8.8 frames per second on a machine where native OpenGL renders the same scene at 36 frames per second.

5.2 Social Network Visualizer

The lightweight code modifications described in Section 3.4 are not very time-consuming, but they do require a fair degree of sophistication from the developer porting the code. However, we present an application that shows how Xax enables developers with no special skill to create new and interesting apps.

We separately ported a Python interpreter and the graphviz graph-layout library to Xax. We also wrote language bindings between Python and the DOM-manipulation functions in JavaScript, which allows Python code to directly manipulate the DOM. Because there are Python wrappers for graphviz, it is possible for Python code to call the powerful graph visualization routines in this library.

Another developer, who had no familiarity with the process of porting code to Xax, then wrote a web app in Python for visualizing a social network, specifically a network of actors from the Internet Movie DataBase (IMDB), akin to "six degrees of Kevin Bacon". The app's web page provides a text box for entering an ac-

tor's name. The client queries a back-end service to enumerate the movies featuring that actor, as well as lists of other actors in each movie. The client-side web app uses graphviz to plot and present the network of actors and movies.

Xax enabled the developer to leverage the advanced non-type-safe legacy code base of grahpviz, which was developed over more than a decade. It also enabled this developer to use his knowledge of Python to write an app without needing to learn a new language, such as JavaScript. None of this required the developer to learn how to port code to Xax. The tool and library were ported once, and they are now usable by non-experts.

5.3 GhostScript and Kaffe

The fact that Xax is yet another browser plugin might give one cause for concern. Even though Xax provides benefits unavailable in any existing plugin, a user might still be bothered by having to install an additional plugin in the browser. However, we present two examples that show how the Xax plugin model can actually subsume other browser plugins.

First, we ported the GhostScript PDF viewer to Xax. PDF viewers are currently available as browser plugins, to enable users to view PDF documents on web sites. Xax enables PDF viewing functionality without the need for a special-purpose plugin. Moreover, by running the PDF viewer inside a secure Xax picoprocess, we protect the browser from the dozens of vulnerabilities that have been discovered in PDF plugins [32].

Second, we performed a basic port of the Kaffe Java Virtual Machine (JVM) into Xax. As described in Section 2.2.1, JVM is an alternative mechanism for writing web applications, albeit one that does not provide legacy support for non-type-safe code. Although we have not yet completed our port of the Qt implementation of Kaffe's Abstract Windowing Toolkit, the core Java execution engine is working and able to perform UI functions via DOM manipulation. As with the PDF renderer, by running the JVM inside a picoprocess, we protect the browser from vulnerabilities in the JVM implementation [4, 9, 10, 37].

6 Evaluation

Here we evaluate Xax's performance, legacy support, OS-independence, and security.

6.1 Performance

To evaluate performance, we run microbenchmarks and macrobenchmarks to measure CPU- and I/O-bound performance. All measurements are on a 2.8GHz Intel Pentium 4.

Xax's use of native CPU execution, adopted to achieve legacy support, also leads to native CPU performance. Our first microbenchmark (Table 1(a)) computes the SHA-1 hash of H.G. Wells' *The War of the Worlds*. Xax performs comparably to the Linux native host. The Windows native binary was compiled with a different compiler (Visual Studio vs. gcc); we believe this explains the improved performance of the Windows native cases.

Environment	tool	compute `sha1` (a)	syscall `close` (b)	alloc `16MB` (c)
Linux native	gcc	5,930,000	430	27,120
Linux Xax	gcc	5,970,000	69,400	202,600
XP native	VS	4,540,000	1,126	31,390
XP Xax	gcc	6,170,000	16,880	235,300
Vista native	VS	4,580,000	1,316	40,900
Vista Xax	gcc	6,490,000	59,900	612,000

Table 1: Microbenchmarks (§6.1). Units are machine cycles, $1/(2.8 \times 10^9)$ sec; max $\frac{\sigma}{\mu}$=6.6%.

The benefits of native execution led us to accept overheads associated with hardware context switching; however, our simple uninvasive user-level implementations lead to quite high overheads. Table 1(b) reports the cost of a null `xaxcall` compared with a null native system call; in each case, we invoke `close(-1)`. Table 1(c) reports the cost of allocating an empty 16MB memory region. The Xax overhead runs 7–161×.

Despite high `xaxcall` overhead, real applications perform quite well because applications use I/O sparingly. Two macrobenchmarks quantify this observation. First, we wrote a Mandelbrot Set viewer that draws 300 distinct 400×400-pixel frames as it zooms into the Set (Table 2(a)). We chose this application for two reasons. First, it involves both intensive computation and frequent I/O. Second, it is small enough to reimplement in multiple languages, enabling us to compare a single application across multiple extension mechanisms. The Mandelbrot benchmark ran at roughly the same speed under both Windows and Linux, under both Xax and using native binaries. For comparison, we also tried two other common browser extension languages: Java and JavaScript. The benchmark ran 30% faster using Sun Java 1.6r7 under Windows XP. This difference arises from the time taken encoding PNG images in the C implementations; to validate this hypothesis, we removed the PNG step, and found the C implementation as fast as Java. Google Chrome's JavaScript engine (build 2200) required more than an hour. We gave up on Internet Explorer 7 and Firefox 3's JavaScript engines after waiting ten minutes for the first two frames.

Second, we compare the performance of rendering and displaying a one-page Postscript document in Xax and on the native host, using Ghostscript 8.62 as the underlying rendering engine in all cases (Table 2(b)). Ghostscript is a rich application that exercises CPU, allocation, and

Environment	Mandelbrot 300 frames (a)	Ghostscript 1 page (b)
Linux native	169s	840ms
Linux Xax	170s	541ms
Win XP native	188s	835ms
Win XP Xax	177s	738ms
Win XP Java	138s	—
JavaScript	4,870s	—

Table 2: Macrobenchmarks.

I/O. The native benchmarks incur an additional overhead of process launch; this test shows only that the Xax performance is reasonable.

6.2 Legacy Support

To evaluate legacy support, we built Xax applications that use 15 libraries totaling 3.3 million lines of code (LoC) in four languages (Table 3). Only minimal changes were needed to compile the libraries. Most changes were configure flags to specify different library dependencies and to emit a static library as output.

6.3 OS-Independence

To evaluate OS-independence, we ran all of the above applications on our Linux 2.6, Windows XP, Windows Vista, and Windows Server 2008 Xax hosts. Figure 2(a) is a Linux-toolchain program running in Firefox on a Linux-PowerPC host. Figure 2(b) is a Windows-toolchain program running in Firefox on Linux-x86. Figure 2(c) is a Linux-toolchain program running in Firefox on Windows XP. Figure 2(d) is a Linux-toolchain program running in Internet Explorer on Windows Vista. Figure 2(e) is a Linux-toolchain program running in Firefox on Linux-x86.

6.4 Security

We roughly compare the strength of different isolation mechanisms by the count of lines of code in their TCBs [6]. The Xax picoprocess TCB is less than 5,000 lines (See Table 4). In contrast, language-based Flash and Java have implementations around two orders of magnitude bigger. Section 2.2.3 considered the alternative of a hardware VM; note that Xen's TCB is similarly large.

The previous comparison is somewhat generous to Xax, because the table counts Kaffe's entire TCB, including both its isolated execution engine (the JVM, around 50,000 lines) and the new native code Kaffe introduces to provide features like rich GUI displays. On the other hand, as a type-safety-based extension mechanism, Kaffe incorporates native UI code in its TCB for performance or to exploit a stack of legacy code. By contrast, Xax Kaffe isolates any native UI code it incorporates. In future work, we expect Xax to isolate Kaffe's

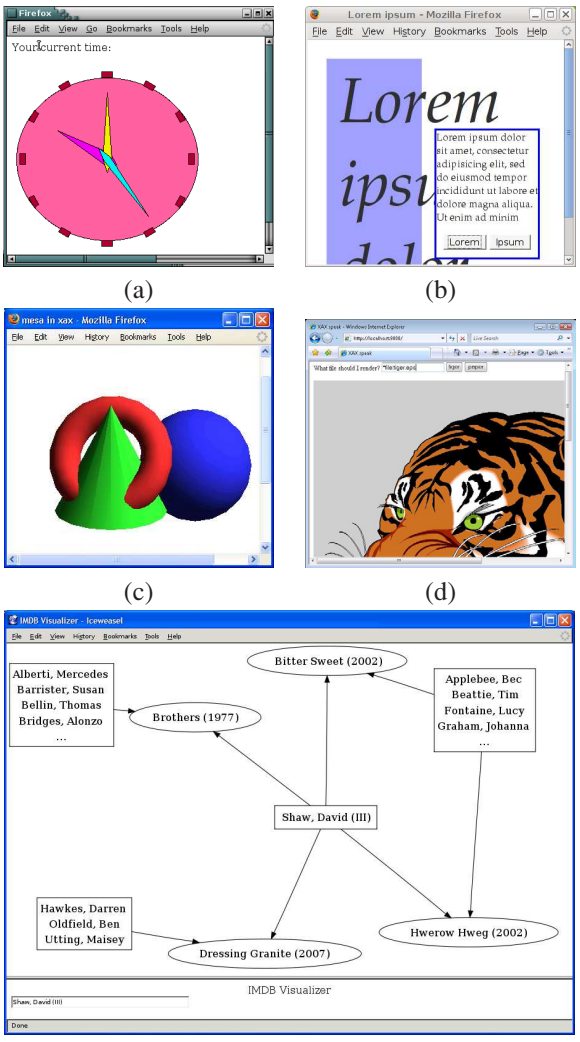

Figure 2: Screenshots of the applications itemized in Table 3.

UI stack with minimal TCB growth. By analogy, note that our PDF viewer isolates a 600K-line component that is otherwise commonly installed in the TCB.

7 Related Work

One of the key observations that enables Xax to achieve its benefits (§2.1) is that today's type-safety-based browser extension mechanisms do not admit legacy code. We chose a simple isolation mechanism buildable from primitives available in commodity operating systems, but several alternative approaches exist.

7.1 Hardware isolation

Other extension systems use memory hardware isolation differently. Nooks isolates drivers from a monolithic kernel [42]. Mondrix has a similar goal, but requires hypothetical hardware support [48]. Palladium isolates kernel extensions by exploiting x86's arcane, otherwise-

Application	Fig.	Language	Toolchain	New LoC	Mods LoC	Base LoC
XaxAnalogClock	2(a)	C	gcc 4.1.2	459		
elfLoader		C	gcc 4.1.2	552		
xaxlib		C	gcc 4.1.2	412		
dietlibc 0.31		C	gcc 4.1.2		405	66,970
zlib 1.2.3		C	gcc 4.1.2		24	25,475
libpng 1.2.25		C	gcc 4.1.2		33	60,925
gd 2.0.35		C	gcc 4.1.2		20	72,202
Kevin Bacon Visualizer	2(e)	Python	python 2.5	143		
xaxFsLib		C	gcc 4.1.2	1,706		
libtar 1.2.11		C	gcc 4.1.2		17	11,348
Python 2.5		C, Python	gcc 4.1.2		33	771,112
Zziplib 0.13.49		C, C++	gcc 4.1.2		40	54,728
Jpeg 6b		C, Asm	gcc 4.1.2		36	26,695
Expat 2.0.1		C	gcc 4.1.2		41	47,813
GraphViz 2.18		C, C++	gcc 4.1.2		79	343,096
Headline Reader		C	gcc 4.1.2	323		
eSpeak 1.36		C++	g++ 4.1.2		46	25,170
3D Demo	2(c)	C, OpenGL	gcc 4.1.2	257		
Mesa 7.0.3		C	gcc 4.1.2		56	683,883
PDF Viewer	2(d)	C	gcc 4.1.2	366		
GhostScript 8.62		C	gcc 4.1.2		29	666,216
Hello Webserver		Java	sun jdk 1.6	67		
Kaffe 1.1.0		C, C++	gcc 4.1.2		123	364,560
Hello Webserver	2(b)	C	Vsl. Studio 2003	189		
peLoader		C	gcc 4.1.2	613		
libc.lib		C			0	57,223
Total					982	3,277,416

Table 3: We have compiled a variety of applications for Xax using a variety of libraries and compiler toolchains comprising over 3.3 million LoC. Each library we build on is listed once, regardless of how many applications use it. For the base libraries, "our LoC" consist mostly of build configuration. LoC is measured by cloc.pl [6].

Isolation mechanism	TCB LoC
Linux `Monitor+Proxy`	2,596
Linux syscall entry path	1,632
	4,228
Windows `Monitor+Proxy`	3,043
Xax kernel driver	978
NT syscall entry path	313
	4,334
Gnash Flash player + deps	791,453
Kaffe non-Java code + deps	280,622
Xen VMM[2]	187,688

Table 4: Security. Because Xax exploits hardware memory protection, both Xax implementations have small TCBs.

unused segment-based isolation mechanism [3]. Each of these mechanisms emphasizes lightweight rich-pointer interactions between a minimally-modified extension and its host environment. Thus the systems improve robustness of existing code compositions, but preventing malice is an explicit non-goal.

7.2 Binary rewriting

Another approach to isolation is binary rewriting. Basic software-fault isolation for RISC [44] and CISC [26] architectures has quite high overhead if required to enforce both read- and write-safety; such mechanisms are thus envisioned as robustness-improving rather than adversary-proof. XFI [12] showed how to achieve adversary-proof isolation for operating system extensions, but overheads still range from 28–116%, and the system has been applied only to short pieces of legacy code.

Vx32 is a general-purpose mechanism that combines segments and binary rewriting [14] to achieve low-overhead adversary-proof protection, and might be a possible alternative to our picoprocesses. However, the use of segment registers places an additional constraint on toolchains. More importantly, it excludes other architectures, which may be important for future mobile devices; Xax already runs on PowerPC (§6.3).

Besides isolation, binary rewriting has also been used to provide transparent cross-architecture portability [2]. It may have applicability in the Xax context: Where web developers have only provided an x86 binary, a non-x86-based host may employ binary rewriting to exchange performance for compatibility.

7.3 Operating system-level virtualization

Another way to isolate untrusted web applications is via *operating system-level virtualization*, wherein an OS provides multiple isolated instances for separate subsystems, each with the same API as the underlying OS. Examples include Solaris zones [33] and FreeBSD jails [41]. The implementation of OS virtualization permeates a monolithic kernel, so its TCB is larger and more amorphous than that of the picoprocess mechanism. Furthermore, the mechanism sacrifices OS-independence and is not supported on many deployed OSes.

7.4 Low-level type safety

Another possible alternative for isolating legacy code is the use of a safety-enforcing typed assembly language (TAL) [29], an instance of a proof-carrying code [31]. TAL is type-safety-based mechanism, similar to those described in §2.2.1, but it is lower-level than JVM or .NET's object-oriented type system. For example, it can enforce safety in polymorphic languages without requiring that objects use vtables. Thus one cannot compile TAL to JVM; a separate TAL runtime must be deployed.

TAL should be easier to target than a higher-level type system; however, no current compilers emit TAL for weakly typed languages such as C. One could perhaps produce such a compiler by modifying the back end of a type-retrofitting C compiler such as CCured [30]. Experience with CCured, however, shows several limitations that restrict its applicability to improving robustness rather than adversary-proofing. First, although some work has been done to verify the safety of CCured's output [19], the CCured compiler must generally be trusted. Second, annotating legacy code to convince CCured to compile it efficiently has proven to be quite labor-intensive [7], and even then rarely eliminates all of the "trusted casts" [19].

7.5 Application context

Many of the papers described above mention potential application to web browsers; however, ours is the first to demonstrate how to enable legacy code to be readily compiled and deployed in the web context. VXA restricts extensions to a narrow interface, and shows applicability to a restricted class of applications (codecs) [13]; one important contribution of the present work is to show how a narrow interface to the existing browser is sufficient to support a much broader range of software (§3.3).

8 Limitations and future work

In this section, we discuss limitations of the current Xax implementation and plans for future enhancements.

8.1 Security analysis

We argue that Xax is secure by its small TCB; however, as a practical matter, our implementations reuse commodity operating systems as substrate. A production implementation deserves a rigorous inspection to ensure both that the kernel syscall dispatch path for a picoprocess is indeed closed, and that no other kernel paths, such as exception handling or memory management, are exploitable by a malicious picoprocess. We should also explore alternative implementations that exclude more host OS code from the TCB, such as a MacOS implementation that uses Mach processes, or a VM-like implementation that completely replaces the processor trap dispatch table for the duration of execution of a picoprocess.

8.2 Rich application enhancements

Rich web applications, Xax or otherwise, will require browser support for efficiently handling large binaries (such as remote differential compression [43]), and support for offline functionality [11, 18]. Because Xax applications access resources via the browser, any browser enhancements that deliver these features are automatically inherited by the Xax environment.

When we port a shared-library loader, Xax can experience further performance improvements from selective preloading [24].

8.3 Improved browser integration

Integrating Xax with the browser using a proxy is expedient, but for several reasons it would be better to directly integrate with the browser. First, rewriting the namespace of the origin server is an abuse of protocol. Instead, the browser should provide an explicit `<embed>` object with which a page can construct and name a picoprocess for further reference. Second, the proxy is unaware of when the browser has navigated away from a page, and when it is thus safe to terminate and reclaim a picoprocess. Third, the proxy cannot operate on `https` connections. For these reasons, we plan to integrate Xax directly into popular browsers.

8.4 Threading

Supporting some threading model is a requirement for targeting general applications. The nonblocking I/O interface (§3.2) is sufficient to implement cooperative user-level threading, such as Kaffe's `jthreads`. Adding to the xaxcall interface a mechanism to deliver an asynchronous signal (e.g., when a poll condition is satisfied) is sufficient to implement preemptive user-level threading. Finally, the xaxcall interface could expose a mechanism for launching additional kernel-level threads to en-

able the picoprocess to exploit a multicore CPU. Each mechanism offers improved application performance in exchange for expanding the Xax Monitor's contribution to the TCB.

8.5 Porting additional libraries

Most of the ports in this paper were built on our modifications to dietlibc. Similarly modifying more mainstream `libc`s, such as the GNU C library and Microsoft Visual Studio's standard libraries, will greatly ease porting of other libraries. One challenge in porting either library is their reliance on x86 segment registers to manage thread-local storage. Because segment registers cannot be assigned in user mode, we must emulate or obviate this functionality.

We also plan to port more interactive code. Our first efforts will be aimed at GUI libraries with few dependencies (e.g. Qt/Embedded). We expect to blit frame buffer regions to the browser; keyboard and mouse events we will capture in JavaScript and send back to Xax.

8.6 Relocatable code

The picoprocess restricts application code to a fixed address range (§3.1). This restriction is an arbitrary intersection of the restrictions imposed by commodity operating systems; we have no guarantee that future Xax implementations will not require further narrowing it. We plan to explore an alternative approach: requiring Xax applications to be relocatable, capable of running on an ABI that makes no guarantee about any particular absolute virtual addresses. One possible limitation with this approach is that it may impede an attempt to import existing binaries directly into Xax.

9 Conclusion

We introduce Xax, a browser plugin model that enables developers to adapt legacy code for use in rich web applications, while maintaining security, performance, and OS-independence.

- Xax's security comes from its use of the picoprocess minimalist isolation boundary and browser-based services; we demonstrate that Xax's TCB is orders of magnitude smaller than alternative approaches.
- Xax's OS-independence comes from its use of picoprocesses and its platform abstraction layer; we demonstrate that Xax applications compiled on any toolchain run on any OS host.
- Xax's performance derives from native code execution in picoprocesses; we measure Xax's compute performance to be comparable with native execution, and that even with quite inefficient I/O performance, Xax delivers compelling whole-application performance.
- Xax's legacy support comes from lightweight code modification; we demonstrate that just a few hundred lines of configuration options are sufficient to port 3.3 million lines of legacy libraries and applications.

Over decades of software development in non-type-safe languages, vast amounts of design, implementation, and testing effort have gone into producing powerful legacy applications. By enabling developers to leverage this prior effort into web applications' deployment and execution model, we anticipate that Xax may change the landscape of web applications.

10 Acknowledgments

The authors thank Jeremy Condit, Chris Hawblitzel, Galen Hunt, Emre Kıcıman, Ed Nightingale, and Helen Wang for enlightening discussions and comments on early drafts of this paper. We also thank the anonymous reviewers and our shepherd, Anthony Joseph, for their suggestions.

References

[1] CARDINAL, C. Live From CES: Hands On With Vista—Vista By The Numbers, A Developer Tells All. Presentation at Gear Live, Jan. 2006.

[2] CHERNOFF, A., AND HOOKWAY, R. DIGITAL FX!32 — running 32-bit x86 applications on Alpha NT. In *Proceedings of the USENIX Windows NT Workshop* (1997), pp. 9–13.

[3] CHIUEH, T., VENKITACHALAM, G., AND PRADHAN, P. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Proceedings of Symposium on Operating Systems Principles (SOSP)* (1999), pp. 140–153.

[4] CIAC M-060. http://www.ciac.org/.

[5] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *Proceedings of Networked Systems Design and Implementation (NSDI)* (2005), pp. 273–286.

[6] CLoC. http://cloc.sourceforge.net/.

[7] CONDIT, J. personal communication, 2008.

[8] COX, R. S., GRIBBLE, S. D., LEVY, H. M., AND HANSEN, J. G. A safety-oriented platform for Web applications. In *Proc. Symposium on Security and Privacy* (2006), pp. 350–364.

[9] CVE-2003-0111. http://cve.mitre.org/.

[10] CVE-2007-0043. http://cve.mitre.org/.

[11] Dojo Toolkit. http://dojotoolkit.org/offline.

[12] ERLINGSSON, U., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. C. XFI: software guards for system address spaces. In *Proceedings of Operating Systems Design and Implementation (OSDI)* (2006), pp. 75–88.

[13] FORD, B. VXA: A virtual architecture for durable compressed archives. In *Proceedings of File and Storage Technologies (FAST)* (2005), pp. 295–308.

[14] FORD, B., AND COX, R. Vx32: Lightweight user-level sandboxing on the x86. In *Proceedings of the USENIX Annual Technical Conference* (2008). To appear.

[15] GARFINKEL, T. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Proceedings of Network and Distributed Systems Security Symposium (NDSS)* (2003), pp. 163–176.

[16] GARFINKEL, T., PFAFF, B., AND ROSENBLUM, M. Ostia: A delegating architecture for secure system call interposition. In *Proc. Network and Distributed Systems Security Symposium* (2004), pp. 187–201.

[17] GOLDBERG, I., WAGNER, D., THOMAS, R., AND BREWER, E. A. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of USENIX Security Symposium* (1996), pp. 1–13.

[18] Google Gears. http://gears.google.com/.

[19] HARREN, M., AND NECULA, G. C. Using dependent types to certify the safety of assembly code. In *Static Analysis Symposium (SAS)* (2005), pp. 155–170.

[20] JACKSON, C., BORTZ, A., BONEH, D., AND MITCHELL, J. Protecting browser state against Web privacy attacks. In *Proceedings of WWW* (2006).

[21] JOBS, S. Keynote address. Apple Worldwide Developers Conference, Aug. 2006.

[22] KICIMAN, E., AND LIVSHITS, B. AjaxScope: A platform for remotely monitoring the client-side behavior of Web 2.0 applications. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)* (2007), ACM.

[23] KOZUCH, M., AND SATYANARAYANAN, M. Internet suspend/resume. In *Proceedings of the Workshop on Mobile Computing Systems and Applications (WMCSA)* (2002), pp. 40–48.

[24] LIVSHITS, B., AND KICIMAN, E. Doloto: Code splitting for network-bound Web 2.0 applications. Tech. Rep. TR 2007-159, 2007.

[25] LORCH, J. R., AND SMITH, A. J. The VTrace tool: building a system tracer for Windows NT and Windows 2000. *MSDN Magazine 15*, 10 (2000), 86–102.

[26] MCCAMANT, S., AND MORRISETT, G. Evaluating SFI for a CISC architecture. In *15th USENIX Security Symposium* (2006), pp. 209–224.

[27] MCILROY, M. D. Mass produced software components. In *Software Engineering*, P. Naur and B. Randell, Eds. NATO Science Committee, Jan. 1969, pp. 138–150.

[28] MICROSOFT CORPORATION. Windows Driver Kit. http://microsoft.com/whdc/devtools/wdk/default.mspx.

[29] MORRISETT, G., WALKER, D., CRARY, K., AND GLEW, N. From System F to typed assembly language. In *Symposium on Principles of Programming Languages (POPL)* (1998), pp. 85–97.

[30] NECULA, G. C., CONDIT, J., HARREN, M., MCPEAK, S., AND WEIMER, W. Ccured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst. 27*, 3 (2005), 477–526.

[31] NECULA, G. C., AND LEE, P. Safe kernel extensions without run-time checking. In *Proceedings of Operating Systems Design and Implementation (OSDI)* (1996), pp. 229–243.

[32] NIST Vulnerability Database. http://nvd.nist.gov/nvd.cfm.

[33] PRICE, D., AND TUCKER, A. Solaris zones: operating system support for server consolidation. In *Proceedings of the 18th Large Installation System Administration Conference (LISA)* (2004), pp. 241–254.

[34] PROVOS, N. Improving host security with system call policies. In *Proceedings of the 12th conference on USENIX Security Symposium (SSYM)* (2003), pp. 18–18.

[35] RINARD, M., CADAR, C., DUMITRAN, D., ROY, D., LEU, T., AND BEEBEE, J. Enhancing server

availability and security through failure-oblivious computing. In *Proceedings of Operating Systems Design and Implementation (OSDI)* (2004), pp. 303–316.

[36] ROBLES, G. Debian counting. http://libresoft.dat.escet.urjc.es/debian-counting/.

[37] SA7587. http://secunia.com/advisories/7587/.

[38] SABIN, T. Strace for NT. http://www.securityfocus.com/tools/1276.

[39] SAPUNTZAKIS, C. P., CHANDRA, R., PFAFF, B., CHOW, J., LAM, M. S., AND ROSENBLUM, M. Optimizing the migration of virtual computers. In *Proceedings of Operating Systems Design and Implementation (OSDI)* (2002), pp. 377–390.

[40] SourceForge. http://sourceforge.net.

[41] STOKELY, M., AND LEE, C. *The FreeBSD Handbook 3rd Edition, Vol. 1: User's Guide*. FreeBSD Mall, Inc., Brentwood, CA, 2003.

[42] SWIFT, M., BERSHAD, B. N., AND LEVY, H. M. Improving the reliability of commodity operating systems. In *Proceedings of Symposium on Operating Systems Principles (SOSP)* (2003), pp. 207–222.

[43] TRIDGELL, A. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, 1999.

[44] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In *Proceedings of Symposium on Operating Systems Principles (SOSP)* (1993), pp. 203–216.

[45] WALDSPURGER, C. A. Memory resource management in VMware ESX server. In *Proceedings of Operating Systems Design and Implementation (OSDI)* (2002), pp. 181–194.

[46] WHITAKER, A., SHAW, M., AND GRIBBLE, S. D. Denali: Lightweight virtual machines for distributed and networked applications. Tech. Rep. 02-02-01, 2002.

[47] Wine Windows compatibility library. www.winehq.org.

[48] WITCHEL, E., RHEE, J., AND ASANOVIC, K. Mondrix: Memory isolation for Linux using Mondriaan memory protection. In *Proceedings of Symposium on Operating Systems Principles (SOSP)* (2005), pp. 31–44.

Notes

[1] The term "Rich Internet Application" or "RIA" is sometimes used to refer to these high-end apps. Since the distinction between a web app and an RIA is fuzzy at best, we consistently use the term "web app" herein.

[2] This value includes code supporting multiple platforms, and thus overestimates the size of the Xen TCB.