JOHN DOUCEUR, JEREMY ELSON, JON HOWELL, AND JACOB R. LORCH, WITH RIK FARROW

leveraging legacy code for Web browsers

John Douceur manages the Distributed Systems Research Group in the Redmond lab of Microsoft Research. His interests are designing algorithms, data structures, and protocols for distributed systems.

johndo@microsoft.com

Jeremy Elson has worked in sensor networks, distributed systems, and occasional hare-brained schemes. He also enjoys flying and likes bicycling to work.

jelson@microsoft.com

Jon Howell works at the intersection of security and scalability in distributed systems. His recent projects focus on the convergence of utility computing and Web-delivered applications.

howell@microsoft.com

Jacob Lorch is a Researcher in the Systems and Networking group at Microsoft Research. His research interests include distributed systems, online games, Web security, and energy management.

lorch@microsoft.com

WEB BROWSERS HAVE BECOME A DE facto user interface for many online applications. But because browser applications are typically written in specialized Web languages, the vast quantity of existing tools, libraries, and applications are unavailable to Web developers. Xax provides a secure execution container that can run legacy code written in arbitrary languages. With a small porting effort, legacy applications can be turned into Xax applications, which execute natively but independently of the underlying OS.

Modern Web applications are driving toward the power of fully functional desktop applications such as email clients (e.g., Gmail, Hotmail, Outlook Web Access) and productivity apps (e.g., 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, the enormous base of legacy code is currently unavailable to the developers of Web applications.

In a paper published at OSDI '08 [1], the authors demonstrated running the GhostScript PDF viewer, the eSpeak speech synthesizer, a Python interpreter, 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 the simple Xax 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.

Xax provides native-code-level performance in a secure and OS-independent manner. Xax relies on 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 via the Xax Monitor
- Lightweight modifications to existing tool chains and code bases, for retargeting legacy code to the Xax picoprocess environment

Picoprocess

Most operating systems rely on hardware memory protection mechanisms to isolate processes from one another. Process isolation prevents one process from interfering with another process by reading or writing its program code or data. But process-level isolation provides insufficient protection for running downloaded code within a browser, as the browser itself is a process owned by the user.

Browsers do run downloaded code, such as JavaScript, Java, and Silverlight, but these are type-safe languages. These languages are interpreted within the browser that enforces a security policy, for example, the Same Origin Policy and limited access to filesystems. Legacy code expects to have complete access to a system via the system call API, and thus it cannot be limited by the browser.

Xax introduces the abstraction of the picoprocess. 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.

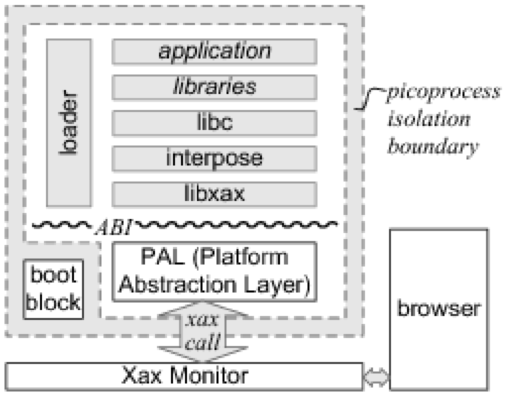

FIGURE 1: THE PICOPROCESS GETS ISOLATED WHEN THE BOOT BLOCK ARRANGES TO INTERCEPT FUTURE SYSTEM CALLS

The Xax Monitor is a user-mode process that creates, isolates, and manages each picoprocess and that provides the functionality of xaxcalls. The Xax Monitor launches the picoprocess, which runs 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.

The boot block is part of the TCB (Trusted Computing Base), even though it executes inside the child process. The boot block uses kernel mechanisms to control access to system calls. The Linux version uses the ptrace() system call, so that all subsequent system calls get trapped and passed to the Xax Monitor. If the Xax Monitor fails (exits), the picoprocess's system calls will no longer be trapped, a weakness in this present Linux implementation. Using ptrace() also hurts performance, as ptrace() was designed for debugging, with the kernel notifying the monitoring process when a system call is made *and* after the system call completes, but before results get passed back to the monitored process.

The Windows version makes a kernel call to establish an interposition on all subsequent syscalls via our XaxDrv driver. 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.

Platform Abstraction Layer

The Platform Abstraction Layer (PAL) 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. Isolation 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).

For memory allocation and deallocation, the ABI provides two calls. The first:

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

maps len zero-filled bytes of picoprocess memory, starting at start if specified, and returns the address. Then:

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

frees the memory region beginning at start, which must be an address returned from xabi_alloc. It returns 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 using this call:

```
int xabi_accept();
```

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

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

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

This returns a channel identifier connected to the given URL, according to the specified method, which may be "get," "put," or "connect." It requests that the Xax Monitor 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. The read and write operations:

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

transfer data on an open channel and return the number of bytes transferred (0 if the channel is not ready, -1 if the channel is closed or failed). The poll operation:

```
int xabi_poll(xabi_poll_fd *pfds, int npfds, bool block);
```

indicates the ready status of a set of channels by updating events. If the value of block is true, it does not return until at least one requested event is ready, thereby allowing the picoprocess to yield the processor. It returns the number of events ready but does not return 0 if the value of block is true. Finally, the close operation:

```
int xabi_close(int chnl);
```

closes an open channel. It returns 0 for success or -1 for error.

During picoprocess boot, the loader needs to know the URL from which to fetch the application image. Xax uses a general loader that reads the application URL from the query parameters of the URL that launched the picoprocess. The following PAL call, which is normally used only by the loader, provides access to these parameters:

```
const char **xabi_args();
```

It returns a pointer to a NULL-terminated list of pointers to arguments specified at instantiation. (Note that there is no corresponding xaxcall; the parameters are written into the PAL during picoprocess initialization.)

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.

A library called libxax exports a set of symbols (xabi_read, xabi_open_url, etc.) that obey the function linkage convention of the developer's tool chain. This shim converts each of these calls to the corresponding ABI call in the PAL. The shim thus provides a standard API to Xax applications.

The Xax Monitor

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 origin 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 origin server could not have done by serving content directly over the Internet. The current Xax Monitor provides this browser interface by acting as a client-side proxy server.

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. The evaluated applications employ a common design pattern: The Xax app sends an HTML page to the browser, and this page contains JavaScript stubs that translate messages from the picoprocess into JavaScript function invocations.

Lightweight Code Modification

Porting legacy applications took surprisingly little effort. This is surprising because the legacy code was written to run atop an operating system, so it was not obvious that the OS-specific code could be eliminated or replaced without crippling the applications. As an example, a quick test using graphviz and a Python interpreter found that this application made 2725 syscalls (39 unique). Porting this code to Xax would seem to require an enormous emulation of OS functionality. However, using lightweight modifications, it was possible to port this code, about a million lines, 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, when ultimately necessary, providing 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, 699 syscalls (16 unique) were eliminated.

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. Restricting the interface is achieved in various ways, such as by setting command-line arguments or environment variables. For Python/graphviz, an entry-point parameter that changes the output method from “xlib” to “svg” was used, eliminating 367 syscalls (21 unique).

The third step is to identify which of the application’s remaining system calls can be handled trivially. In some cases, it is adequate to return error codes indicating failure, in a manner similar to failure-oblivious computing [2]. For Python/graphviz, it was sufficient to simply reject 125 syscalls (11 unique: 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 interposition layer (see Figure 1). For instance, Python/graphviz reads Python library files from a file system at runtime. The authors packaged these library files as a tarball and emulated a subset of filesystem calls using libtar to access the libraries. The tarball is read-only, which is all Python/graphviz requires. For some of the other ported applications, the authors also provided read/write access to temporary files by creating a RAM disk in the interposition layer. Code in the interposition layer looks at the file path to determine whether to direct calls to the tarball, to the RAM disk, or to somewhere else, such as a file downloaded from the origin server. For Python/graphviz, they used 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 get routed to the UI in the browser. The authors provided this functionality for the remaining 137 syscalls (11 unique: 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, much of the syscall support developed for one app can be readily reused for other apps. The internally emulated tar-based file system was written to support eSpeak and later reused to support Python. Similarly, the backing functionality for the mmap functions and networking functions (listen, accept, bind . . .) are used by all of the 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, BSD, and Win32 syscall interfaces.

Performance

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

Xax's use of native CPU execution, adopted to achieve legacy support, also leads to native CPU performance. The first microbenchmark [Table 1, column (a)] computed the SHA-1 hash of H.G. Wells's *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 versus gcc), likely producing the improved performance of the Windows native cases over Xax.

Environment	Tool	Computation	Syscall	Allocation
		SHA-1	close	16 MB
		(a)	(b)	(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 IN UNITS OF MACHINE CYCLES, 1/(2.8X10^9) SEC; MAX [(SIGMA)/(MU)] = 6.6%

The benefits of native execution allowed the authors to accept overheads associated with hardware context switching. However, the simple noninvasive user-level implementations lead to quite high overheads. Table 1, column (b) reports the cost of a null xaxcall compared with a null native system call (close(-1)). Table 1, column (c) reports the cost of allocating an empty 16-MB memory region. The Xax overhead runs 7–161x.

Limitations and Future Work

For related work, we refer you to Section 7 of the OSDI paper [1]. In terms of security, the authors argue that Xax is secure by its small TCB. However, 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. The authors suggest exploring alternative implementations that exclude more host OS code from the TCB, such as a Mac OS 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.

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

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, Xax currently rewrites the namespace of the origin server; this 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, the authors plan to integrate Xax directly into popular browsers.

Other issues include supporting conventional threading models, using a more mainstream C library such as glibc (here dietlibc was used), and using relocatable code (since Xax uses statically linked code).

Conclusions

Xax is a browser plug-in 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; 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; Xax applications can be compiled on any toolchain and run on any OS host.
- Xax's performance derives from native code execution in picoprocesses.
- Xax's legacy support comes from lightweight code modification.

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 a Web application deployment and execution model, we anticipate that Xax may change the landscape of Web applications.

REFERENCES

[1] John R. Douceur, Jeremy Elson, Jon Howell, and Jacob R. Lorch, "Leveraging Legacy Code to Deploy Desktop Applications on the Web": http://www.usenix.org/events/osdi08/tech/full_papers/douceur/douceur_html/index.html.

[2] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, T. Leu, and W.S. Beebee, Jr., "Enhancing Server Availability and Security through Failure-Oblivious Computing": http://www.usenix.org/events/osdi04/tech/rinard.html.