Published in Proceedings of 2008 USENIX Annual Technical Conference

Vx32: Lightweight User-level Sandboxing on the x86

Bryan Ford and Russ Cox
Massachusetts Institute of Technology
{*baford,rsc*}*@pdos.csail.mit.edu*

Abstract

Code sandboxing is useful for many purposes, but most sandboxing techniques require kernel modifications, do not completely isolate guest code, or incur substantial performance costs. Vx32 is a multipurpose user-level sandbox that enables any application to load and safely execute one or more guest plug-ins, confining each guest to a system call API controlled by the host application and to a restricted memory region within the host's address space. Vx32 runs guest code efficiently on several widespread operating systems without kernel extensions or special privileges; it protects the host program from both reads and writes by its guests; and it allows the host to restrict the instruction set available to guests. The key to vx32's combination of portability, flexibility, and efficiency is its use of x86 segmentation hardware to sandbox the guest's data accesses, along with a lightweight instruction translator to sandbox guest instructions.

We evaluate vx32 using microbenchmarks and whole system benchmarks, and we examine four applications based on vx32: an archival storage system, an extensible public-key infrastructure, an experimental user-level operating system running atop another host OS, and a Linux system call jail. The first three applications export custom APIs independent of the host OS to their guests, making their plug-ins binary-portable across host systems. Compute-intensive workloads for the first two applications exhibit between a 30% slowdown and a 30% *speedup* on vx32 relative to native execution; speedups result from vx32's instruction translator improving the cache locality of guest code. The experimental user-level operating system allows the use of the guest OS's applications alongside the host's native applications and runs faster than whole-system virtual machine monitors such as VMware and QEMU. The Linux system call jail incurs up to 80% overhead but requires no kernel modifications and is delegation-based, avoiding concurrency vulnerabilities present in other interposition mechanisms.

1 Introduction

A *sandbox* is a mechanism by which a *host* software system may execute arbitrary *guest* code in a confined environment, so that the guest code cannot compromise or affect the host other than according to a well-defined policy. Sandboxing is useful for many purposes, such as running untrusted Web applets within a browser [6], safely extending operating system kernels [5, 32], and limiting potential damage caused by compromised applications [19, 22]. Most sandboxing mechanisms, however, either require guest code to be (re-)written in a type-safe language [5, 6], depend on special OS-specific facilities [8, 15, 18, 19], allow guest code unrestricted read access to the host's state [29, 42], or entail a substantial performance cost [33, 34, 37].

Vx32 is a lightweight sandbox for the x86 architecture that enables applications to run untrusted code efficiently on standard operating systems without requiring special privileges or kernel extensions. The vx32 sandbox runs standard x86 instructions, so guest code may be written in any language including assembly language, and may use advanced processor features such as vector (SSE) instructions. An application may host multiple sandbox instances at once; vx32 gives each guest its own dynamically movable and resizable address space within the host's space. Vx32 confines both guest reads and guest writes to the guest's designated address region in the host, protecting both the host's integrity and the privacy of any sensitive data (e.g., SSL keys) it may hold in its address space. Vx32 confines each guest's system calls to an API completely determined by the host application. The guest system call API need not have any relationship to that of the host operating system, so the host application can keep its guest environments independent of and portable across host operating systems.

The key to vx32's combination of flexibility and efficiency is to use different mechanisms to sandbox data accesses and instruction execution. Vx32 sandboxes guest

data accesses using the x86 processor's segmentation hardware, by loading a special data segment into the `ds`, `es`, and `ss` registers before executing guest code. Accessing data through this segment automatically confines both reads and writes to the guest's designated address region, with no performance overhead since the processor always performs segment translation anyway.

Since the vx32 sandbox runs entirely in user mode, however, vx32 cannot rely on the processor's privilege level mechanism to prevent the guest from escaping its sandbox—for example, the x86 privilege levels alone would not prevent the guest from changing the segment registers. Vx32 therefore prevents guest code from executing "unsafe" instructions such as segment register loads by using dynamic instruction translation [9, 34], rewriting each guest code sequence into a "safe" form before executing it. This dynamic translation incurs some performance penalty, especially on control flow instructions, which vx32 must rewrite to keep execution confined to its cache of safe, rewritten code. Since vx32 confines data accesses via segmentation, it does not need to rewrite most computation instructions, leaving safe code sequences as compact and efficient as the guest's original code. Vx32's on-demand translation can in fact improve the cache locality of the guest code, sometimes resulting in better performance than the original code, as seen previously in dynamic optimization systems [4].

Because common OS kernels already provide user-level access to the x86 segmentation hardware, vx32 does not require any special privileges or kernel extensions in order to fully sandbox all memory reads and writes that guest code performs.

Vx32 is implemented as a library that runs on Linux, FreeBSD, and Mac OS X and is being used in several applications. VXA [13] is an archival storage system that stores executable decoders along with compressed content in archives, using vx32 to run these decoders at extraction time; thus the archives are "self-extracting" but also safe and OS-independent. Alpaca [24] is an extensible PKI framework based on proof-carrying authorization [3] that uses vx32 to execute cryptographic algorithms such as SHA-1 [12] that form components of untrusted PKI extensions. Plan 9 VX is a port of the Plan 9 operating system [35] to user space: Plan 9 kernel code runs as a user-level process atop another OS, and unmodified Plan 9 user applications run under the Plan 9 kernel's control inside vx32. Vxlinux is a delegation-based system call interposition tool for Linux. All of these applications rely on vx32 to provide near-native performance: if an extension mechanism incurs substantial slowdown, then in practice most users will forego extensibility in favor of faster but less flexible schemes.

Previous papers on VXA [13] and Alpaca [24] briefly introduced and evaluated vx32 in the context of those applications. This paper focuses on the vx32 virtual machine itself, describing its sandboxing technique in detail and analyzing its performance over a variety of applications, host operating systems, and hardware. On real applications, vx32 consistently executes guest code within a factor of two of native performance; often the overhead is just a few percent.

This paper first describes background and related work in Section 2, then presents the design of vx32 in Section 3. Section 4 evaluates vx32 on its own, then Section 5 evaluates vx32 in the context of the above four applications, and Section 6 concludes.

2 Related Work

Many experimental operating system architectures permit one user process to isolate and confine others to enforce a "principle of least privilege": examples include capability systems [25], L3's clan/chief model [26], Fluke's nested process architecture [14], and generic software wrappers [15]. The primary performance cost of kernel-mediated sandboxes like these is that of traversing hardware protection domains, though with careful design this cost can be minimized [27]. Other systems permit the kernel itself to be extended with untrusted code, via domain-specific languages [31], type-safe languages [5], proof-carrying code [32], or special kernel-space protection mechanisms [40]. The main challenge in all of these approaches is deploying a new operating system architecture and migrating applications to it.

Other work has retrofitted existing kernels with sandboxing mechanisms for user processes, even taking advantage of x86 segments much as vx32 does [8]. These mechanisms still require kernel modifications, however, which are not easily portable even between different x86-based OSes. In contrast, vx32 operates entirely in user space and is easily portable to any operating system that provides standard features described in Section 3.

System call interposition, a sandboxing method implemented by Janus [19] and similar systems [7, 17, 18, 22, 36], requires minor modifications to existing kernels to provide a means for one user process to filter or handle selected system calls made by another process. Since the sandboxed process's system calls are still fielded by the host OS before being redirected to the user-level "supervisor" process, system call interposition assumes that the sandboxed process uses the same basic system call API as the host OS: the supervisor process cannot efficiently export a completely different (e.g., OS-independent) API to the sandboxed process as a vx32 host application can. Some system call interposition methods also have concurrency-related security vulnerabilities [16, 43], whose only clear solution is delegation-based interposition [17]. Although vx32 has other uses,

it can be used is to implement efficient delegation-based system call interposition, as described in Section 5.4.

Virtualization has been in use for decades for purposes such as sharing resources [10] and migrating applications to new operating systems [20]. Since the x86 architecture did not provide explicit support for virtualization until recently, x86-based virtual machines such as VMware [1] had to use dynamic instruction translation to run guest kernel code in an unprivileged environment while simulating the appearance of being run in privileged mode: the dynamic translator rewrites instructions that might reveal the current privilege level. Virtual machines usually do not translate user-mode guest code, relying instead on host kernel extensions to run user-mode guest code directly in a suitably constructed execution environment. As described in Section 5.3, vx32's dynamic translation can be used to construct virtual machines that need no host kernel extensions, at some performance cost.

Dynamic instruction translation is frequently used for purposes other than sandboxing, such as dynamic optimization [4], emulating other hardware platforms [9, 44] or code instrumentation and debugging [28, 34]. The latter two uses require much more complex code transformations than vx32 performs, with a correspondingly larger performance cost [37].

A software fault isolation (SFI) system [29, 42] statically transforms guest code, preprocessing it to create a specialized version in which it is easy for the verifier to check that all data write instructions write only to a designated "guest" address range, and that control transfer instructions branch only to "safe" code entrypoints. SFI originally assumed a RISC architecture [42], but PittSFIeld adapted SFI to the x86 architecture [29]. SFI's preprocessing eliminates the need for dynamic instruction translation at runtime but increases program code size: e.g., 60%-100% for PittSFIeld. For efficiency, SFI implementations typically sandbox only writes and branches, not reads, so the guest can freely examine host code and data. This may be unacceptable if the host application holds sensitive data such as passwords or SSL keys. The main challenge in SFI on x86 is the architecture's variable-length instructions: opcode sequences representing unsafe instructions might appear in the middle of legitimate, safe instructions. PittSFIeld addresses this problem by inserting no-ops so that all branch targets are 16-byte aligned and then ensures that branches clear the bottom four bits of the target address. MiSFIT [39] sidesteps this problem for direct jumps by loading only code that was assembled and cryptographically signed by a trusted assembler. Indirect jumps consult a hash table listing valid jump targets.

Applications can use type-safe languages such as Java [6] or C# [30] to implement sandboxing completely in user space. This approach requires guest code to be written in a particular language, making it difficult to reuse existing legacy code or use advanced processor features such as vector instructions (SSE) to improve the performance of compute-intensive code.

3 The Vx32 Virtual Machine

The vx32 virtual machine separates data sandboxing from code sandboxing, using different, complementary mechanisms for each: x86 segmentation hardware to sandbox data references and dynamic instruction translation to sandbox code. The dynamic instruction translation prevents malicious guest code from escaping the data sandbox. Vx32's dynamic translation is simple and lightweight, rewriting only indirect branches and replacing unsafe instructions with virtual traps. The use of dynamic translation also makes it possible for client libraries to restrict the instruction set further.

This section describes the requirements that vx32 places on its context—the processor, operating system, and guest code—and then explains the vx32 design.

3.1 Requirements

Processor architecture. Vx32 is designed around the x86 architecture, making the assumption that most systems now and in the foreseeable future are either x86-based or will be able to emulate x86 code efficiently. This assumption appears reasonable in the current desktop and server computing market, although it may prevent vx32 from spreading easily into other domains, such as game consoles and handheld mobile devices.

Vx32 uses protected-mode segmentation, which has been integral to the x86 architecture since before its extension to 32 bits [21]. The recent 64-bit extension of the architecture disables segment translation in 64-bit code, but still provides segmentation for 32-bit code [2]. Vx32 therefore cannot use segmentation-based data sandboxing to run 64-bit guest code, but it can still run 32-bit sandboxed guest code within a 64-bit host application.

Host operating system. Vx32 requires that the host OS provide a method of inserting custom segment descriptors into the application's local descriptor table (LDT), as explained below. The host OS can easily and safely provide this service to all applications, provided it checks and restricts the privileges of custom segments. All widely-used x86 operating systems have this feature.[1]

To catch and isolate exceptions caused by guest code, vx32 needs to register its own signal handlers for processor exceptions such as segmentation faults and floating point exceptions. For full functionality and robustness, the host OS must allow vx32 to handle these signals on a

[1] One Windows vulnerability, MS04-011, was caused by inadequate checks on application-provided LDT segments: this was merely a bug in the OS and not an issue with custom segments in general.

separate signal stack, passing vx32 the full saved register state when such a signal occurs. Again, all widely-used x86 operating systems have this capability.

Finally, vx32 can benefit from being able to map disk files into the host application's address space and to control the read/write/execute permissions on individual pages in the mapping. Although these features are not strictly required by vx32, they are, once again, provided by all widely-used x86 operating systems.

On modern Unix variants such as Linux, FreeBSD, and OS X, specific system calls satisfying the above requirements are `modify_ldt`/`i386_set_ldt`, `sigaction`, `sigaltstack`, `mmap`, and `mprotect`. Windows NT, 2000, and XP support equivalent system calls, though we have not ported vx32 to Windows. We have not examined whether Windows Vista retains this functionality.

Guest code. Although vx32 uses x86 segmentation for data sandboxing, it assumes that guest code running in the sandbox conforms to the 32-bit "flat model" and makes no explicit reference to segment registers. In fact, vx32 rewrites any guest instructions referring to segment registers so that they raise a virtual illegal instruction exception. This "flat model" assumption is reasonable for practically all modern, compiled 32-bit x86 code; it would typically be a problem only if, for example, the sandboxed guest wished to run 16-bit DOS or Windows code or wished to run a nested instance of vx32 itself.

Some modern multithreading libraries use segment registers to provide quick access to thread-local storage (TLS); such libraries cannot be used in guest code under the current version of vx32, but this is not a fundamental limitation of the approach. Vx32 could be enhanced to allow guest code to create new segments using emulation techniques, perhaps at some performance cost.

Host applications may impose further restrictions on guest code through configuration flags that direct vx32 to reject specific classes of instructions. For example, for consistent behavior across processor implementations, the VXA archiver described in Section 5.1 disallows the non-deterministic 387 floating-point instructions, forcing applications to use deterministic SSE-based equivalents.

3.2 Data sandboxing: segmentation

In the x86 architecture, segmentation is an address translation step that the processor applies immediately before page translation. In addition to the eight general-purpose registers (GPRs) accessible in user mode, the processor provides six *segment registers*. During any memory access, the processor uses the value in one of these segment registers as an index into one of two segment translation tables, the *global descriptor table* (GDT) or *local descriptor table* (LDT). The GDT traditionally describes segments shared by all processes, while the LDT contains segments specific to a particular process. Upon finding the appropriate descriptor table entry, the processor checks permission bits (read, write, and execute) and compares the virtual address of the requested memory access against the *segment limit* in the descriptor table, throwing an exception if any of these checks fail. Finally, the processor adds the *segment base* to the virtual address to form the *linear address* that it subsequently uses for page translation. Thus, a normal segment with base b and limit l permits memory accesses at virtual addresses between 0 and l, and maps these virtual addresses to linear addresses from b to $b+l$. Today's x86 operating systems typically make segmentation translation a no-op by using a base of 0 and a limit of $2^{32}-1$. Even in this so-called "flat model," the processor continues to perform segmentation translation: it cannot be disabled.

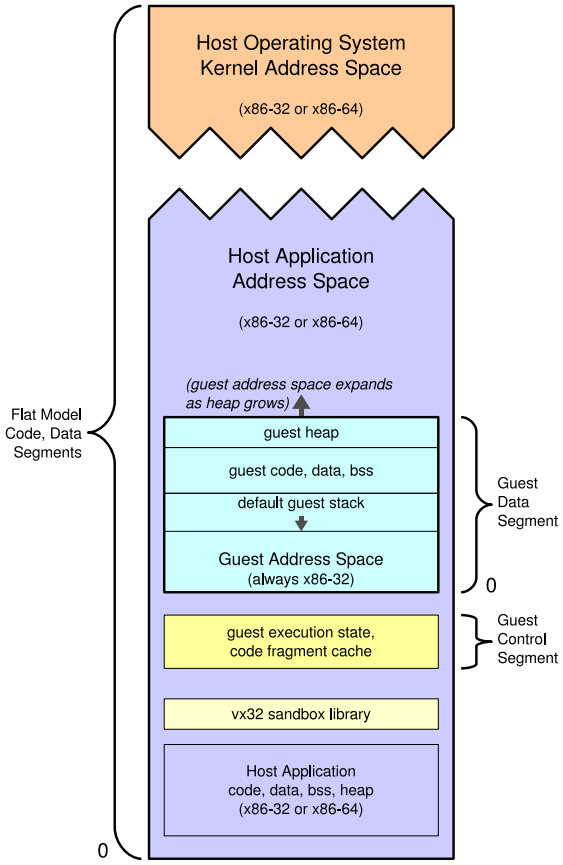

Figure 1: Guest and Host Address Space Structure

Vx32 allocates two segments in the host application's LDT for each guest instance: a *guest data segment* and a *guest control segment*, as depicted in Figure 1.

The *guest data segment* corresponds exactly to the guest instance's address space: the segment base points to the beginning of the address space (address 0 in the guest instance), and the segment size is the guest's address space size. Vx32 executes guest code with the processor's `ds`, `es`, and `ss` registers holding the selec-

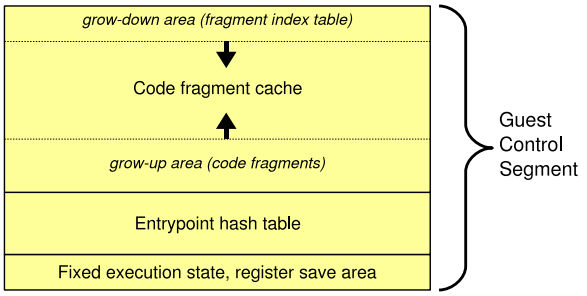

Figure 2: Guest Control Segment Structure

tor for the guest data segment, so that data reads and writes performed by the guest access this segment by default. (Code sandboxing, described below, ensures that guest code cannot override this default.) The segmentation hardware ensures that the address space appears at address 0 in the guest and that the guest cannot access addresses past the end of the segment. The translation also makes it possible for the host to unmap a guest's address space when it is not in use and remap it later at a different host address, to relieve congestion in the host's address space for example.

The format of the guest data segment is up to vx32's client: vx32 only requires that it be a contiguous, page-aligned range of virtual memory within the host address space. Vx32 provides a loader for ELF executables [41], but clients can load guests by other means. For example, Plan 9 VX (see section 5.3) uses `mmap` and `mprotect` to implement demand loading of Plan 9 executables.

The *guest control segment*, shown in Figure 2, contains the data needed by vx32 during guest execution. The segment begins with a fixed data structure containing saved host registers and other data. The *entrypoint hash table* and *code fragment cache* make up most of the segment. The hash table maps guest virtual addresses to code sequences in the code fragment cache. The translated code itself needs to be included in the guest control segment so that vx32 can write to it when patching previously-translated unconditional branches to jump directly to their targets [38].

Vx32 executes guest code with the processor's `fs` or `gs` register holding the selector for the guest control segment. The vx32 runtime accesses the control segment by specifying a segment override on its data access instructions. Whether vx32 uses `fs` or `gs` depends on the host system, as described in the next section.

3.3 Code sandboxing: dynamic translation

Data sandboxing ensures that, using the proper segments, data reads and writes cannot escape the guest's address space. Guests could still escape using segment override prefixes or segment register loads, however, which are unprivileged x86 operations. Vx32 therefore uses code scanning and dynamic translation to prevent guest code from performing such unsafe operations.

As in Valgrind [34] and just-in-time compilation [11, 23], vx32's code scanning and translation is fully dynamic and runs on demand. The guest is allowed to place arbitrary code sequences in its address space, but vx32 never executes this potentially-unsafe code directly. Instead, whenever vx32 enters a guest instance, it translates a fragment of code starting at the guest's current instruction pointer (`eip`) to produce an equivalent safe fragment in vx32's code fragment cache, which lies *outside* the guest's address space. Vx32 also records the `eip` and address of the translated fragment in the entrypoint hash table for reuse if the guest branches to that `eip` again. Finally, vx32 jumps to the translated code fragment; after executing, the fragment either returns control to vx32 or jumps directly to the next translated fragment.

On 32-bit hosts, vx32 never changes the code segment register (`cs`): it jumps directly to the appropriate fragment in the guest's code fragment cache. This is safe because the code fragment cache only contains safe translations generated by vx32 itself. The code translator ensures that all branches inside translated code only jump to the beginning of other translated fragments or back to vx32 to handle events like indirect branches or virtualized guest system calls.

On 64-bit hosts, since segmentation only operates while executing 32-bit code, vx32 must create a special 32-bit code segment mapping the low 4GB of the host address space for use when running guest code. The guest control and data segments must therefore reside in the low 4GB of the host address space on such systems, although other host code and data may be above 4GB.

Because vx32 never executes code in the guest's address space directly, vx32 requires no static preprocessing or verification of guest code before it is loaded, in contrast with most other sandboxing techniques. Indeed, reliably performing static preprocessing and verification is problematic on the x86 due to the architecture's variable-length instructions [29, 39].

Translation overview. Vx32's translation of guest code into code fragments is a simple procedure with four stages: scan, simplify, place, and emit. The stages share a "hint table" containing information about each instruction in the fragment being translated. The eventual output is both the translated code and the hint table, which the translator saves for later use by exception handlers.

1. *Scan*. The translator first scans guest code starting at the desired `eip`, decoding x86 instructions to determine their lengths and any required transformations. The translator scans forward until it reaches an unconditional branch or a fragment size limit (currently about 128 bytes of instructions). The

scan phase records the length, original offset, instruction type, and worst-case translated size in the hint table. Jumps are the only instructions whose translated size is not known exactly at this point.

2. *Simplify*. The next phase scans the hint table for direct branches within the fragment being translated; it marks the ones that can be translated into short intrafragment branches using 8-bit jump offsets. After this phase, the hint table contains the exact size of the translation for each original guest instruction.

3. *Place*. Using the now-exact hint table information, the translator computes the exact offset of each instruction's translation. These offsets are needed to emit intrafragment branches in the last phase.

4. *Emit*. The final phase writes the translation into the code fragment cache. For most instructions, the translation is merely a copy of the original instruction; for "unsafe" guest instructions, the translation is an appropriate sequence chosen by vx32.

Vx32 saves the hint table, at a cost of four bytes per original instruction, in the code fragment cache alongside each translation, for use in exception handling as described in Section 3.4. The hint table could be discarded and recomputed during exception handling, trading exception handling performance for code cache space.

The rest of this section discusses specific types of guest instructions. Figure 3 shows concrete examples.

Computational code. Translation leaves most instructions intact. All ordinary computation and data access instructions (`add`, `mov`, and so on) and even floating-point and vector instructions are "safe" from vx32's perspective, requiring no translation, because the segmentation hardware checks all data reads and writes performed by these instructions against the guest data segment's limit. The only computation instructions that vx32 does not permit the guest to perform directly are those with x86 segment override prefixes, which change the segment register used to interpret memory addresses and could thus be used to escape the data sandbox.

Guest code may freely use all eight general-purpose registers provided by the x86 architecture: vx32 avoids both the dynamic register renaming and spilling of translation engines like Valgrind [34] and the static register usage restrictions of SFI [42]. Allowing guest code to use all the registers presents a practical challenge for vx32, however: it leaves no general-purpose register available where vx32 can store the address of the saved host registers for use while entering or exiting guest code. As mentioned above, vx32 solves this problem by placing the information in the guest control segment and using an otherwise-unused segment register (`fs` or `gs`) to address it. (Although vx32 does not permit segment override prefixes in guest code, it is free to insert them for its own use in the code fragment translations.)

It is common nowadays for thread libraries to use one of these two segment registers—`fs` or `gs`—as a pointer to thread-local storage. If vx32 reused the thread-local segment register, it would have to restore the segment register before calling any thread-aware library routines, including routines that perform locking, such as `printf`. On recent GCC-based systems, the thread-local segment register is even used in function call prologues to look up the stack limit during a stack overflow check. Also, some 64-bit x86 operating systems (e.g., Linux) use privileged instructions to initialize the thread-local segment register with a base that is impossible to represent in an ordinary 32-bit segment descriptor. On such systems, restoring the thread-local segment register would require a system call, increasing the cost of exiting guest code. For these reasons, vx32 uses whichever segment register is not being used by the host OS's thread library. With care, vx32 could share the thread library's segment register.

Control transfers. To keep guest execution safely confined to its cache of translated code fragments, vx32 must ensure that all control transfer instructions—calls, jumps, and returns—go to vx32-generated translations, not to the original, unsafe guest code.

In the worst case, a control transfer must search the translation hash table, invoking the instruction translator if no translation exists. Once a translation has been found, vx32 can rewrite or "patch" direct jumps and direct calls to avoid future lookups [34, 38]. To implement this patching, the instruction translator initially translates each fixed-target jump or call instruction to jump to a stub that invokes the hash table lookup and branch patching function. The branch patching function looks up the target address and then rewrites the jump or call instruction to transfer directly to the target translation.

Patching cannot be used for indirect branches, including indirect calls and returns. This hash table lookup for indirect branches, especially during return instructions, is the main source of slowdown in vx32.

Other dynamic translation systems optimize indirect branches by caching the last target of each indirect branch and the corresponding translation address, or by maintaining a cache of subroutine return targets analogous to what many modern processors do [37]. Such optimizations would be unlikely to benefit vx32: its indirect target lookup path is only 21 instructions in the common case of an immediate hash table hit. Only the computation of the hash index—5 instructions—would be eliminated by using a single-entry branch cache. Most of the other instructions, which save and restore the x86 condition code flags and a few guest registers to give the target lookup code "room to work," would still be required no matter how simple the lookup itself.

(a) An indirect jump to the address stored at 08049248:

```
08048160  jmp   [0x08049248]
          ⇓
b7d8d0f9  mov   ebx, fs:[0x2c]
b7d8d100  mov   fs:[0x2c], ebx
b7d8d107  mov   ebx, [0x08049248]
b7d8d10d  jmp   vxrun_lookup_indirect
```

The `fs` segment register points to the guest control segment. The first line of *every* translated code fragment is a prologue that restores the guest's `ebx` (at `b7d8d0f9` in this case), because vx32 jumps into a fragment using a `jmp [ebx]` instruction.

The translation of the `jmp` instruction itself begins on the second line (at `b7d8d100`). The translated code saves `ebx` back into the guest control segment, loads the target `eip` into `ebx`, and then jumps to `vxrun_lookup_indirect`, which locates and jumps to the cached fragment for the guest address in `ebx`.

The first two lines cannot be optimized out: other fragments may directly jump past the first instruction, as shown below.

(b) A direct jump to 08048080:

```
08048160  jmp   0x08048080
          ⇓
b7d8d0f9  mov   ebx, fs:[0x2c]
b7d8d100  jmp   0xb7d8d105
b7d8d105  mov   fs:[0x5c], 0x00008115
b7d8d110  jmp   vxrun_lookup_backpatch
b7d8d115  dword 0x08048080
b7d8d119  dword 0xb7d8d105
```

The first `jmp` in the translation is initially a no-op that just jumps to the next instruction, but `vxrun_lookup_backpatch` will rewrite it to avoid subsequent lookups. The word stored into `fs:[0x5c]` is an `fs`-relative offset telling `vxrun_lookup_backpatch` where in the control segment to find the two dwords arguments at `b7d8d115`. The control segment for the guest begins at `b7d85000` in this example.

The first argument is the target `eip`; the second is the address of the end of the 32-bit jump offset to be patched. Since `ebx` has not been spilled at the point of the jump, `vxrun_lookup_backpatch` patches the jump to skip the one-instruction prologue in the target fragment that restores `ebx`.

(c) A return instruction:

```
08048160  ret
          ⇓
b7d8d0f9  mov   ebx, fs:[0x2c]
b7d8d100  mov   fs:[0x2c], ebx
b7d8d107  pop   ebx
b7d8d108  jmp   vxrun_lookup_indirect
```

A return is an indirect jump to an address popped off the stack.

(d) An indirect call:

```
08048160  call  [0x08049248]
          ⇓
b7d8d0f9  mov   ebx, fs:[0x2c]
b7d8d100  mov   fs:[0x2c], ebx
b7d8d107  mov   ebx, [0x08049248]
b7d8d10d  push  0x08048166
b7d8d112  jmp   vxrun_lookup_indirect
```

The translation is almost identical to the one in (a). The added `push` instruction saves the guest return address onto the stack.

(e) A direct call:

```
08048160  call  0x8048080
          ⇓
b7d8d0f9  mov   ebx, fs:[0x2c]
b7d8d100  push  0x8048165
b7d8d105  jmp   0xb7d8d10a
b7d8d10a  mov   fs:[0x5c], 0x0000811a
b7d8d115  jmp   vxrun_lookup_backpatch
b7d8d11a  dword 0x08048080
b7d8d11e  dword 0xb7d8d10a
```

The translation is identical to the one in (b) except for the addition of the `push` that saves the return address.

(f) A software interrupt:

```
08048160  int   0x30
          ⇓
b7d8d0f9  mov   ebx, fs:[0x2c]
b7d8d100  mov   fs:[0x20], eax
b7d8d106  mov   eax, 0x230
b7d8d10b  mov   fs:[0x40], 0x8048162
b7d8d116  jmp   vxrun_gentrap
```

The translation saves the guest `eax` into the guest control segment, loads the virtual trap number into `eax` (the `0x200` bit indicates an `int` instruction), saves the next `eip` into the guest control segment, and then jumps to the virtual trap handler, which will stop the execution loop and return from vx32, letting the library's caller handle the trap.

(g) An unsafe or illegal instruction:

```
08048160  mov   ds, ax
          ⇓
b7d8d0f9  mov   ebx, fs:[0x2c]
b7d8d100  mov   fs:[0x20], eax
b7d8d106  mov   eax, 0x006
b7d8d10b  mov   fs:[0x40], 0x8048160
b7d8d116  jmp   vxrun_gentrap
```

The translation generates a virtual trap with code `0x006`. In contrast with (f), for illegal instructions the saved `eip` points at the guest instruction itself rather than just past it.

Figure 3: *Guest code* and vx32 translations. Most instructions—arithmetic, data moves, and so on—are unchanged by translation.

Traps. Vx32 translates instructions like `int`, `syscall`, and `sysenter`, which normally generate hardware traps, into code sequences that generate virtual traps instead: they record the trap code and then cause vx32 to return to its caller, allowing the host application to handle the trap as it wishes. Typical applications look for a specific trap code to interpret as a "virtual system call" and treat any other trap as reason to terminate the guest.

Privileged or unsafe instructions. Vx32 translates privileged or unsafe instructions (for example, kernel-mode instructions or those user-mode instructions that manipulate the segment registers) into sequences that generate (virtual) illegal instruction traps.

3.4 Exception handling

With help from the host OS, vx32 catches processor exceptions in guest code—for example, segmentation violations and floating point exceptions—and turns them into virtual traps, returning control to the host application with full information about the exception that occurred.

Since the `eip` reported by the host OS on such an exception points into one of vx32's code translations, vx32 must translate this `eip` back to the corresponding `eip` in the guest's original instruction stream in order for it to make sense to the host application or the developer. To recover this information, vx32 first locates the translation fragment containing the current `eip` and converts the `eip`'s offset within the fragment to an offset from the guest code address corresponding to the fragment.

To locate the translation fragment containing the trapping `eip` efficiently, vx32 organizes the code fragment cache into two sections as shown earlier in Figure 2: the code translations and instruction offset tables are allocated from the bottom up, and the fragment index is allocated from the top down. The top-down portion of the cache is thus a table of all the translation fragments, sorted in reverse order by fragment address. The exception handler uses a binary search in this table to find the fragment containing a particular `eip` as well as the hint table constructed during translation.

Once vx32's exception handler has located the correct fragment, it performs a second binary search, this one in the fragment's hint table, to find the exact address of the guest instruction corresponding to the current `eip`.

Once the exception handler has translated the faulting `eip`, it can finally copy the other guest registers unchanged and exit the guest execution loop, transferring control back to the host application to handle the fault.

3.5 Usage

Vx32 is a generic virtual execution library; applications decide how to use it. Typically, applications use vx32 to execute guest code in a simple control loop: load a register set into the vx32 instance, and call vx32's `run` function; when `run` eventually returns a virtual trap code, handle the virtual trap; repeat. Diversity in vx32 applications arises from what meaning they assign to these traps. Section 5 describes a variety of vx32 applications and evaluates vx32 in those contexts.

Vx32 allows the creation of multiple guest contexts that can be run independently. In a multithreaded host application, different host threads can run different guest contexts simultaneously with no interference.

4 Vx32 Evaluation

This section evaluates vx32 in isolation, comparing vx32's execution against native execution through microbenchmarks and whole-system benchmarks. Section 5 evaluates vx32 in the context of real applications. Both sections present experiments run on a variety of test machines, listed in Figure 4.

4.1 Implementation complexity

The vx32 sandbox library consists of 3,800 lines of C (1,500 semicolons) and 500 lines of x86 assembly language. The code translator makes up about half of the C code. Vx32 runs on Linux, FreeBSD, and Mac OS X without kernel modifications or access to privileged operating system features.

In addition to the library itself, the vx32 system provides a GNU compiler toolchain and a BSD-derived C library for optional use by guests hosted by applications that provide a Unix-like system call interface. Host applications are, of course, free to use their own compilers and libraries and to design new system call interfaces.

4.2 Microbenchmarks

To understand vx32's performance costs, we wrote a small suite of microbenchmarks exercising illustrative cases. Figure 5 shows vx32's performance on these tests.

Jump. This benchmark repeats a sequence of 100 noop short jumps. Because a short jump is only two bytes, the targets are only aligned on 2-byte boundaries. In contrast, vx32's generated fragments are aligned on 4-byte boundaries. The processors we tested vary in how sensitive they are to jump alignment, but almost all run considerably faster on vx32's 4-byte aligned jumps than the 2-byte jumps in the native code. The Pentium 4 and the Xeon are unaffected.

Jumpal. This benchmark repeats a sequence of 100 short jumps that are spaced so that each jump target is aligned on a 16-byte boundary. Most processors execute vx32's equivalent 4-byte aligned jumps a little slower. The Pentium 4 and Xeon are, again, unaffected.

Jumpfar. This benchmark repeats a sequence of 100 jumps spaced so that each jump target is aligned on a 4096-byte (page) boundary. This is a particularly hard

Label		CPU(s)	RAM	Operating System
Athlon64 x86-32		1.0GHz AMD Athlon64 2800+	2GB	Ubuntu 7.10, Linux 2.6.22 (32-bit)
Core 2 Duo	1x2	2.33GHz Intel Core 2 Duo	1GB	Mac OS X 10.4.10
Opteron x86-32		1.4GHz AMD Opteron 240	1GB	Ubuntu 7.10, Linux 2.6.22 (32-bit)
Opteron x86-64		1.4GHz AMD Opteron 240	1GB	Ubuntu 7.10, Linux 2.6.22 (64-bit)
Pentium 4		3.06GHz Intel Pentium 4	2GB	Ubuntu 7.10, Linux 2.6.22
Pentium M		1.0GHz Intel Pentium M	1GB	Ubuntu 7.04, Linux 2.6.10
Xeon	2x2	3.06GHz Intel Xeon	2GB	Debian 3.1, Linux 2.6.18

Figure 4: Systems used during vx32 evaluation. The two Opteron listings are a single machine running different operating systems. The notation 1x2 indicates a single-processor machine with two cores. All benchmarks used gcc 4.1.2.

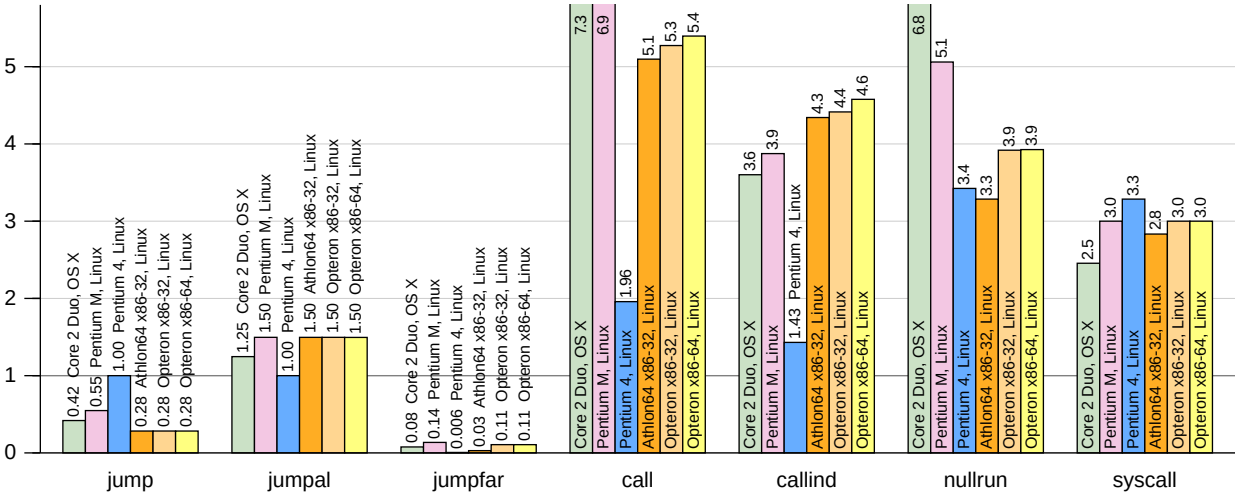

Figure 5: Normalized run times for microbenchmarks running under vx32. Each bar plots run time using vx32 divided by run time for the same benchmark running natively (smaller bars mark faster vx32 runs). The benchmarks are described in Section 4.2. Results for the Intel Xeon matched the Pentium 4 almost exactly and are omitted for space reasons.

case for native execution, especially if the processor's instruction cache uses only the low 12 bits of the instruction address as the cache index. Vx32 runs this case significantly faster on all processors, because of better instruction cache performance in the translation.

Call. This benchmark repeatedly calls a function containing only a return instruction. The call is a direct branch, though the return is still an indirect branch.

Callind. This benchmark is the same as *call*, but the call is now an indirect branch, via a register.

Comparing the bars for *call* against the bars for *callind* may suggest that vx32 takes longer to execute direct function calls than indirect function calls, but only relative to the underlying hardware: a vx32 indirect call takes about twice as long as a vx32 direct call, while a native indirect call takes about four times as long as a native direct call. The *call* bars are taller than the *callind* bars not because vx32 executes direct calls more slowly, but because native hardware executes them so much faster.

Nullrun. This benchmark compares creating and executing a vx32 guest instance that immediately exits against forking a host process that immediately exits.

Syscall. This benchmark compares a virtual system call relayed to the host system against the same system call executed natively. (The system call is `close(-1)`, which should be trivial for the OS to execute.)

4.3 Large-scale benchmarks

The microbenchmarks help to characterize vx32's performance executing particular kinds of instructions, but the execution of real programs depends critically on how often the expensive instructions occur. To test vx32 on real programs, we wrote a 500-line host application called vxrun that loads ELF binaries [41] compiled for a generic Unix-like system call interface. The system call interface is complete enough to support the SPEC CPU2006 integer benchmark programs, which we ran both using vx32 (vxrun) and natively. We ran only the C integer benchmarks; we excluded 403.gcc and 429.mcf because they caused our test machines, most of which have only 1GB of RAM, to swap.

Figure 6 shows the performance of vx32 compared to the native system on five different 32-bit x86 processors. On three of the seven benchmarks, vx32 incurs a perfor-

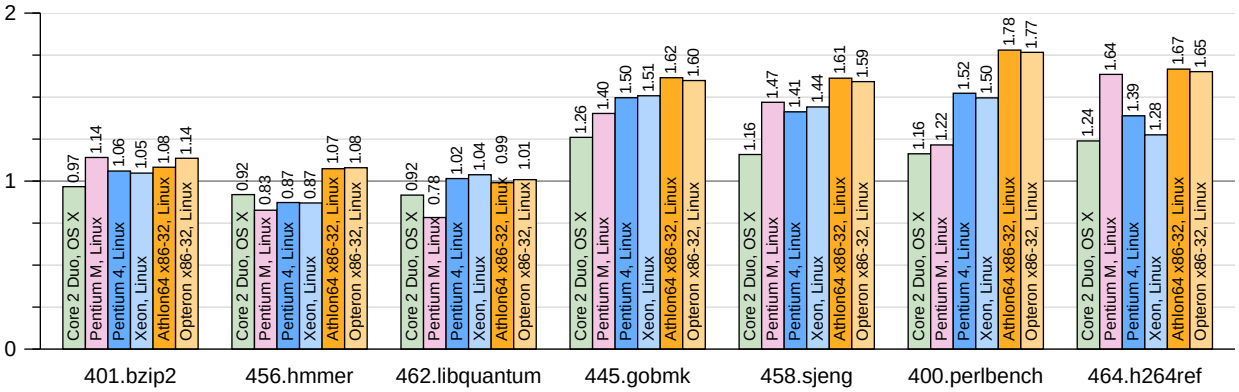

Figure 6: Normalized run times for SPEC CPU2006 benchmarks running under vx32. Each bar plots run time using vx32 divided by run time for the same benchmark running natively (smaller bars mark faster vx32 runs). The left three benchmarks use fewer indirect branches than the right four, resulting in less vx32 overhead. The results are discussed further in Section 4.3.

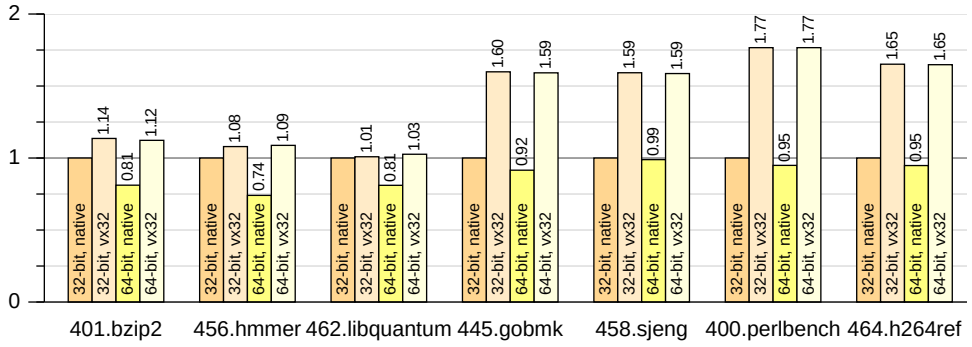

Figure 7: Normalized run times for SPEC CPU2006 benchmarks running in four configurations on the same AMD Opteron system: natively on 32-bit Linux, under vx32 hosted by 32-bit Linux, natively on 64-bit Linux, and under vx32 hosted by 64-bit Linux. Each bar plots run time divided by run time for the same benchmark running natively on 32-bit Linux (smaller bars mark faster runs). Vx32 performance is independent of the host operating system's choice of processor mode, because vx32 always runs guest code in 32-bit mode. The results are discussed further in Section 4.3.

mance penalty of less than 10%, yet on the other four, the penalty is 50% or more. The difference between these two groups is the relative frequency of indirect branches, which, as discussed in Section 3, are the most expensive kind of instruction that vx32 must handle.

Figure 8 shows the percentage of indirect branches retired by our Pentium 4 system during each SPEC benchmark, obtained via the CPU's performance counters [21]. The benchmarks that exhibit a high percentage of indirect call, jump, and return instructions are precisely those that suffer a high performance penalty under vx32.

We also examined vx32's performance running under a 32-bit host operating system compared to a 64-bit host operating system. Figure 7 graphs the results. Even under a 64-bit operating system, the processor switches to 32-bit mode when executing vx32's 32-bit code segments, so vx32's execution time is essentially identical in each case. Native 64-bit performance often differs from 32-bit performance, however: the x86-64 architecture's eight additional general-purpose registers can improve performance by requiring less register spilling in compiled code, but its larger pointer size can hurt performance by decreasing cache locality, and the balance between these factors depends on the workload.

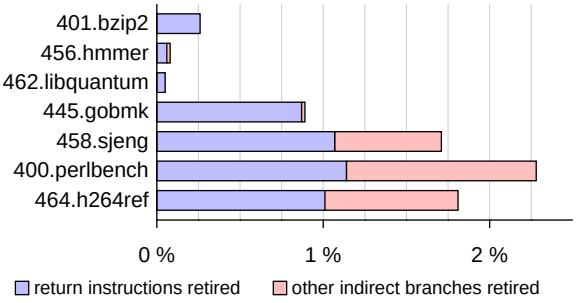

Figure 8: Indirect branches as a percentage of total instructions retired during SPEC CPU2006 benchmarks, measured using performance counters on the Pentium 4. The left portion of each bar corresponds to return instructions; the right portion corresponds to indirect jumps and indirect calls. The indirect-heavy workloads are exactly those that experience noticeable slowdowns under vx32.

5 Applications

In addition to evaluating vx32 in isolation, we evaluated vx32 in the context of several applications built using it. This section evaluates the performance of these applications, but equally important is the ability to create them in the first place: vx32 makes it possible to create interesting new applications that execute untrusted x86 code on legacy operating systems without kernel modifications, at only a modest performance cost.

5.1 Archival storage

VXA [13] is an archival storage system that uses vx32 to "future proof" compressed data archives against changes in data compression formats. Data compression algorithms evolve much more rapidly than processor architectures, so VXA packages executable decoders into the compressed archives along with the compressed data itself. Unpacking the archive in the future then depends only on being able to run on (or simulate) an x86 processor, not on having the original codecs used to compress the data and being able to run them natively on the latest operating systems. Crucially, archival storage systems need to be efficiently usable now as well as in the future: if "future proofing" an archive using sandboxed decoders costs too much performance in the short term, the archive system is unlikely to be used except by professional archivists.

VXA uses vx32 to implement a minimal system call API (`read`, `write`, `exit`, `sbrk`). Vx32 provides exactly what the archiver needs: it protects the host from buggy or malicious archives, it isolates the decoders from the host's system call API so that archives are portable across operating systems and OS versions, and it executes decoders efficiently enough that VXA can be used as a general-purpose archival storage system without noticeable slowdown. To ensure that VXA decoders behave identically on all platforms, VXA instructs vx32 to disable inexact instructions like the 387 intrinsics whose precise results vary from one processor to another; VXA decoders simply use SSE and math library equivalents.

Figure 9 shows the performance of vx32-based decoders compared to native ones on the four test architectures. All run within 30% of native performance, often much closer. The jpeg decoder is consistently faster under vx32 than natively, due to better cache locality.

5.2 Extensible public key infrastructure

Alpaca [24] is an extensible public-key infrastructure (PKI) and authorization framework built on the idea of proof-carrying authorization (PCA) [3], in which one party authenticates itself to another by using an explicit logical language to *prove* that it deserves a particular kind of access or is authorized to request particular services. PCA systems before Alpaca assumed a fixed set of cryptographic algorithms, such as public-key encryption, signature, and hash algorithms. Alpaca moves these algorithms into the logical language itself, so that the extensibility of PCA extends not just to delegation policy but also to complete cryptographic suites and certificate formats. Unfortunately, cryptographic algorithms like round-based hash functions are inefficient to express and evaluate explicitly using Alpaca's proof language.

Alpaca uses Python bindings for the vx32 sandbox to support native implementations of expensive algorithms like hashes, which run as untrusted "plug-ins" that are fully isolated from the host system. The lightweight sandboxing vx32 provides is again crucial to the application, because an extensible public-key infrastructure is unlikely to be used in practice if it makes all cryptographic operations orders of magnitude slower than native implementations would be.

Figure 10 shows the performance of vx32-based hash functions compared to native ones. All run within 25% of native performance. One surprise is the Core 2 Duo's excellent performance, especially on whirlpool. We believe the Core 2 Duo is especially sensitive to cache locality.

5.3 Plan 9 VX

Plan 9 VX (9vx for short) is a port of the Plan 9 operating system [35] to run on top of commodity operating systems, allowing the use of both Plan 9 and the host system simultaneously and also avoiding the need to write hardware drivers. To run user programs, 9vx creates an appropriate address space in a window within its own address space and invokes vx32 to simulate user mode execution. Where a real kernel would execute `iret` to enter user mode and wait for the processor to trap back into kernel mode, 9vx invokes vx32 to simulate user mode, waiting for it to return with a virtual trap code. 9vx uses a temporary file as a simulation of physical memory, calling the host `mmap` and `mprotect` system calls to map individual memory pages as needed. This architecture makes it possible to simulate Plan 9's shared-memory semantics exactly, so that standard Plan 9 x86 binaries run unmodified under 9vx. For example, Plan 9 threads have a shared address space except that each has a private stack. This behavior is foreign to other systems and very hard to simulate directly. Because all user-mode execution happens via vx32, 9vx can implement this easily with appropriate memory mappings.

The most surprising aspect of 9vx's implementation was how few changes it required. Besides removing the hardware drivers, it required writing about 1,000 lines of code to interface with vx32, and another 500 to interface with the underlying host operating system. The changes mainly have to do with page faults. 9vx treats vx32 like an architecture with a software-managed TLB (the code

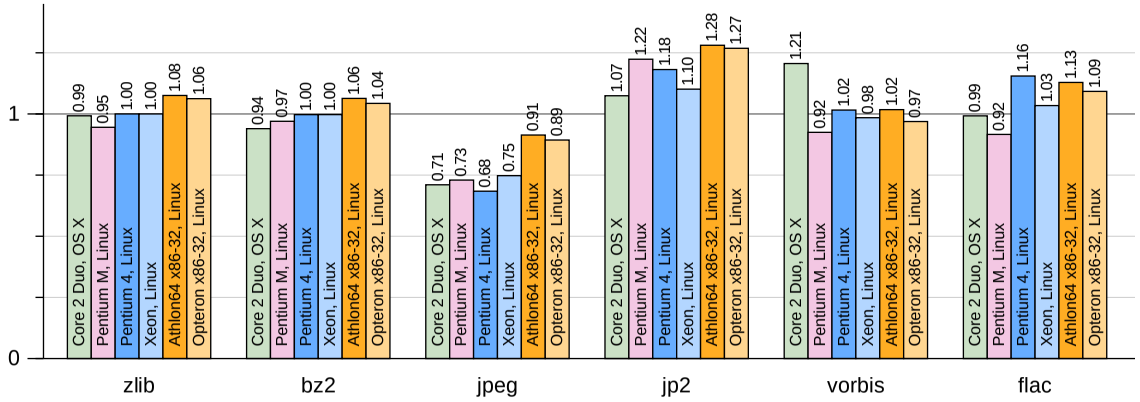

Figure 9: Normalized run times for VXA decoders running under vx32. Each bar plots run time using vx32 divided by run time for the same benchmark running natively (smaller bars mark faster vx32 runs). Section 5.1 gives more details. The jpeg test runs faster because the vx32 translation has better cache locality than the original code.

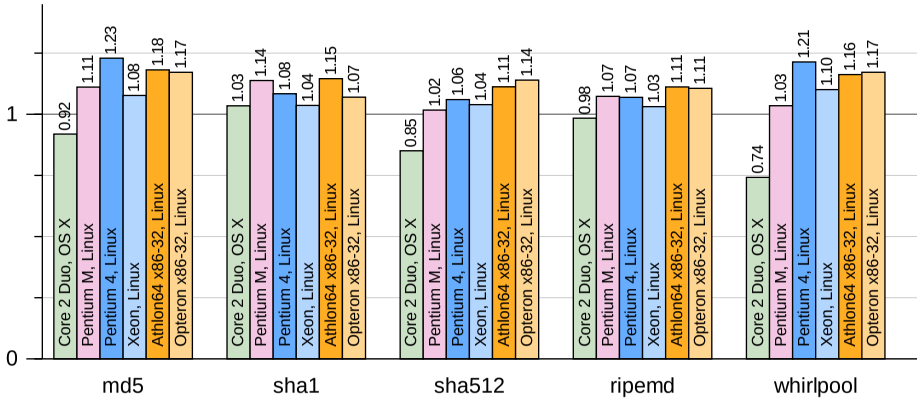

Figure 10: Normalized run times for cryptographic hash functions running under vx32. Each bar plots run time using vx32 divided by run time for the same benchmark running natively (smaller bars mark faster runs).

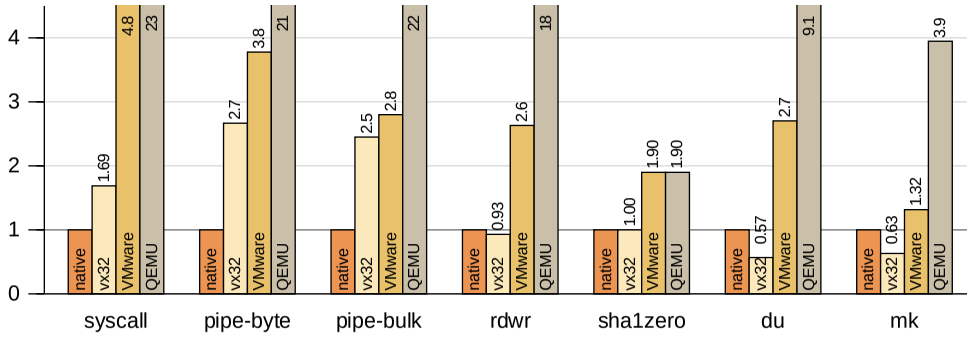

Figure 11: Normalized run times for simple Plan 9 benchmarks. The four bars correspond to Plan 9 running natively, Plan 9 VX, Plan 9 under VMware Workstation 6.0.2 on Linux, and Plan 9 under QEMU on Linux using the `kqemu` kernel extension. Each bar plots run time divided by the native Plan 9 run time (smaller bars mark faster runs). The tests are: swtch, a system call that reschedules the current process, causing a context switch (`sleep(0)`); pipe-byte, two processes sending a single byte back and forth over a pair of pipes; pipe-bulk, two processes (one sender, one receiver) transferring bulk data over a pipe; rdwr, a single process copying from `/dev/zero` to `/dev/null`; sha1zero, a single process reading `/dev/zero` and computing its SHA1 hash; du, a single process traversing the file system; and mk, building a Plan 9 kernel. See Section 5.3 for performance explanations.

was already present in Plan 9 to support architectures like the MIPS). 9vx unmaps all mapped pages during a process context switch (a single `munmap` call) and then remaps pages on demand during vx32 execution. A fault on a missing page causes the host kernel to send 9vx a signal (most often `SIGSEGV`), which causes vx32 to stop and return a virtual trap. 9vx handles the fault exactly as Plan 9 would and then passes control back to vx32. 9vx preempts user processes by asking the host OS to deliver `SIGALRM` signals at regular intervals; vx32 translates these signals into virtual clock interrupts.

To evaluate the performance of 9vx, we ran benchmarks on our Pentium M system in four configurations: native Plan 9, 9vx on Linux, Plan 9 under VMware Workstation 6.0.2 (build 59824) on Linux, and Plan 9 under QEMU on Linux with the `kqemu` module. Figure 11 shows the results. 9vx is slower than Plan 9 at context switching, so switch-heavy workloads suffer (swtch, pipe-byte, pipe-bulk). System calls that don't context switch (rdwr) and ordinary computation (sha1zero) run at full speed under 9vx. In fact, 9vx's simulation of system calls is faster than VMware's and QEMU's, because it doesn't require simulating the processor's entry into and exit from kernel mode. File system access (du, mk) is also faster under 9vx than Plan 9, because 9vx uses Linux's in-kernel file system while the other setups use Plan 9's user-level file server. User-level file servers are particularly expensive in VMware and QEMU due to the extra context switches. We have not tested Plan 9 under VMware ESX server, which could be more efficient than VMware Workstation since it bypasses the host OS completely.

The new functionality 9vx creates is more important than its performance. Using vx32 means that 9vx requires no special kernel support to make it possible to run Plan 9 programs and native Unix programs side-by-side, sharing the same resources. This makes it easy to experiment with and use Plan 9's features while avoiding the need to maintain hardware drivers and port large pieces of software (such as web browsers) to Plan 9.

5.4 Vxlinux

We implemented a 250-line host application, vxlinux, that provides delegation-based interposition [17] by running unmodified, single-threaded Linux binaries under vx32 and relaying the guest's system calls to the host OS. A complete interposition system would include a policy controlling which system calls to relay, but for now we merely wish to evaluate the basic interposition mechanism. The benefit of vxlinux over the OS-independent vxrun (described in Section 4) is that it runs unmodified Linux binaries without requiring recompilation for vx32. The downside is that since it implements system calls by passing arguments through to the Linux kernel, it can only run on Linux. The performance of the SPEC benchmarks under vxlinux is essentially the same as the performance under vxrun; we omit the graph.

6 Conclusion

Vx32 is a multipurpose user-level sandbox that enables any application to load and safely execute one or more guest plug-ins, confining each guest to a system call API controlled by the host application and to a restricted memory region within the host's address space. It executes sandboxed code efficiently on x86 architecture machines by using the x86's segmentation hardware to isolate memory accesses along with dynamic code translation to disallow unsafe instructions.

Vx32's ability to sandbox untrusted code efficiently has enabled a variety of interesting applications: self-extracting archival storage, extensible public-key infrastructure, a user-level operating system, and portable or restricted execution environments. Because vx32 works on widely-used x86 operating systems without kernel modifications, these applications are easy to deploy.

In the context of these applications (and also on the SPEC CPU2006 benchmark suite), vx32 always delivers sandboxed execution performance within a factor of two of native execution. Many programs execute within 10% of the performance of native execution, and some programs execute faster under vx32 than natively.

Acknowledgments

Chris Lesniewski-Laas is the primary author of Alpaca. We thank Austin Clements, Stephen McCamant, and the anonymous reviewers for valuable feedback. This research is sponsored by the T-Party Project, a joint research program between MIT and Quanta Computer Inc., Taiwan, and by the National Science Foundation under FIND project 0627065 (User Information Architecture).

References

[1] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS XIII*, December 2006.

[2] Advanced Micro Devices, Inc. AMD x86-64 architecture programmer's manual, September 2002.

[3] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *6th ACM CCS*, November 1999.

[4] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[5] Brian N. Bershad et al. Extensibility, safety and performance in the SPIN operating system. In *15th SOSP*, 1995.

[6] Brian Case. Implementing the Java virtual machine. *Microprocessor Report*, 10(4):12–17, March 1996.

[7] Suresh N. Chari and Pau-Chen Cheng. BlueBox: A policy-driven, host-based intrusion detection system. In *Network and Distributed System Security*, February 2002.

[8] Tzi-cker Chiueh, Ganesh Venkitachalam, and Prashant Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *17th SOSP*, pages 140–153, December 1999.

[9] Bob Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. *SIGMETRICS PER*, 22(1):128–137, May 1994.

[10] R. J. Creasy. The origin of the VM/370 time-sharing system. *IBM Journal of Research and Development*, 25(5):483–490, 1981.

[11] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Principles of Programming Languages*, pages 297–302, Salt Lake City, UT, January 1984.

[12] D. Eastlake 3rd and T. Hansen. US secure hash algorithms (SHA and HMAC-SHA), July 2006. RFC 4634.

[13] Bryan Ford. VXA: A virtual architecture for durable compressed archives. In *4th USENIX FAST*, San Francisco, CA, December 2005.

[14] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. Microkernels meet recursive virtual machines. In *2nd OSDI*, pages 137–151, 1996.

[15] Timothy Fraser, Lee Badger, and Mark Feldman. Hardening COTS software with generic software wrappers. In *IEEE Symposium on Security and Privacy*, pages 2–16, 1999.

[16] Tal Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Network and Distributed System Security*, February 2003.

[17] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *Network and Distributed System Security*, February 2004.

[18] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, and Thomas E. Anderson. SLIC: An extensibility system for commodity operating systems. In *USENIX*, June 1998.

[19] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *6th USENIX Security Symposium*, San Jose, CA, 1996.

[20] Honeywell Inc. *GCOS Environment Simulator*. December 1983. Order Number AN05-02A.

[21] Intel Corporation. IA-32 Intel architecture software developer's manual, June 2005.

[22] K. Jain and R. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *Network and Distributed System Security*, February 2000.

[23] Andreas Krall. Efficient JavaVM just-in-time compilation. In *Parallel Architectures and Compilation Techniques*, pages 54–61, Paris, France, October 1998.

[24] Christopher Lesniewski-Laas, Bryan Ford, Jacob Strauss, M. Frans Kaashoek, and Robert Morris. Alpaca: extensible authorization for distributed services. In *ACM Computer and Communications Security*, October 2007.

[25] Henry M Levy. *Capability-based Computer Systems*. Digital Press, 1984.

[26] Jochen Liedtke. A persistent system in real use: experiences of the first 13 years. In *IWOOOS*, 1993.

[27] Jochen Liedtke. On micro-kernel construction. In *15th SOSP*, 1995.

[28] Chi-Keung Luk et al. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, June 2005.

[29] Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC architecture. In *15th USENIX Security Symposium*, August 2006.

[30] Microsoft Corporation. C# language specification, version 3.0, 2007.

[31] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Symposium on Operating System Principles*, pages 39–51, Austin, TX, November 1987.

[32] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *2nd OSDI*, pages 229–243, 1996.

[33] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. In *Third Workshop on Runtime Verification (RV'03)*, Boulder, CO, July 2003.

[34] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, June 2007.

[35] Rob Pike et al. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[36] Niels Provos. Improving host security with system call policies. In *12th USENIX Security Symposium*, August 2003.

[37] K. Scott et al. Overhead reduction techniques for software dynamic translation. In *NSF Workshop on Next Generation Software*, April 2004.

[38] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, 1993.

[39] Christopher Small and Margo Seltzer. MiSFIT: Constructing safe extensible systems. *IEEE Concurrency*, 6(3):34–41, 1998.

[40] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *19th ACM SOSP*, 2003.

[41] Tool Interface Standard (TIS) Committee. Executable and linking format (ELF) specification, May 1995.

[42] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

[43] Robert N. M. Watson. Exploiting concurrency vulnerabilities in system call wrappers. In *1st USENIX Workshop on Offensive Technologies*, August 2007.

[44] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Measurement and Modeling of Computer Systems*, pages 68–79, 1996.