Capsicum: practical capabilities for UNIX

Robert N. M. Watson[*] and Jonathan Anderson[†]
University of Cambridge
Computer Laboratory
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
{robert.watson, jonathan.anderson}@cl.cam.ac.uk

Ben Laurie and Kris Kennaway
Google
Belgrave House
76 Buckingham Palace Road
London SW1W 9TQ
United Kingdom
{benl, kennaway}@google.com

5 February, 2010

Abstract

Capsicum is a lightweight operating system capability and sandbox framework planned for inclusion in FreeBSD 9. Capsicum extends, rather than replaces, UNIX APIs, providing new kernel primitives (sandboxed *capability mode* and *capabilities*) and a userspace sandbox API. These tools support the compartmentalization of monolithic UNIX applications into logical applications. We demonstrate our approach by adapting core FreeBSD utilities and Google's Chromium web browser to use Capsicum primitives, and compare the complexity and robustness of Capsicum with other sandboxing techniques.

1 Introduction

Capsicum is an API that brings capabilities to UNIX[1]. Capabilities are unforgeable tokens of authority, and have long been the province of research operating systems such as PSOS [16] and EROS [23]. UNIX systems have less fine-grained access control than capability systems, but are very widely deployed. By adding capability primitives to standard UNIX APIs, Capsicum gives application authors a realistic adoption path towards one of the ideals of OS security: least-privilege operation.

Today, many popular, security-critical applications have been decomposed into parts with different privilege requirements. Privilege separation, or *compartmentalisation*, is a pattern that has been adopted by applications such as OpenSSH, Apple's SecurityServer, and, more recently, Google's Chromium web browser. Compartmentalisation is enforced using various existing access control techniques, but only with significant programmer and computational effort; current OS facilities are simply not designed for this purpose.

The two systems of access control in conventional (non-capability-oriented) operating systems are *Discretionary Access Control* (DAC) and *Mandatory Access Control* (MAC). DAC was designed to protect users from each other: the owner of an object (such as a file) can specify *permissions* for it, which are checked by the OS whenever the object is accessed. MAC was designed to enforce system policies: system administrators specify policies (e.g. "users cleared to Secret may not read Top Secret documents"), which are checked via run-time hooks inserted into many places in the operating system's kernel.

[*]Generously supported by a grant from Google, Inc.

[†]Generously supported by the Rothermere Foundation and the Natural Sciences and Research Council of Canada.

[1]We have built a Capsicum prototype, which is open source and planned for inclusion in FreeBSD 9. Further information and source code are available at `http://www.cl.cam.ac.uk/research/security/capsicum/`.

Neither of these systems was designed to address the case of a single application processing many types of information on behalf of one user. For instance, a modern web browser must parse HTML, scripting languages, images and video from many untrusted sources, but also has full access to the user's home directory (this privilege is known as *ambient authority*).

In order to protect user data from malicious JavaScript, Flash, etc., the Chromium web browser is decomposed into several OS processes. Some of these processes handle content from untrusted sources, but their access to user data is restricted (the process is *sandboxed*). The DAC and MAC mechanisms used vary from platform to platform, but all of them require a significant amount of programmer effort (from hundreds of lines of code or policy to, in one case, 22,000 lines of C++) and, sometimes, UNIX root privilege, yet all are vulnerable to simple attacks (see Section 5).

We have modified several applications, including base FreeBSD utilities and Chromium, to use Capsicum primitives. No special privileges are required, and code changes are minimal: the `tcpdump` utility, which has been plagued with security vulnerabilities in the past, can be sandboxed with Capsicum in around ten lines of code, and Chromium can have OS-supported sandboxing in just 100 lines.

In addition to being more secure and easier to use than other sandboxing techniques, Capsicum is highly performant: unlike pure capability systems where system calls necessarily employ message passing, Capsicum's capability-aware system calls are just a few percent slower than their UNIX counterparts, and the `gzip` utility incurs a constant-time penalty of 2.4 ms for the security of a Capsicum sandbox (see Section 6).

2 Capsicum design

Capsicum is designed to blend capabilities with UNIX. This approach achieves many of the benefits of least-privilege operation, while preserving existing UNIX APIs and performance, presenting application authors with a viable adoption path for capability-oriented software architecture.

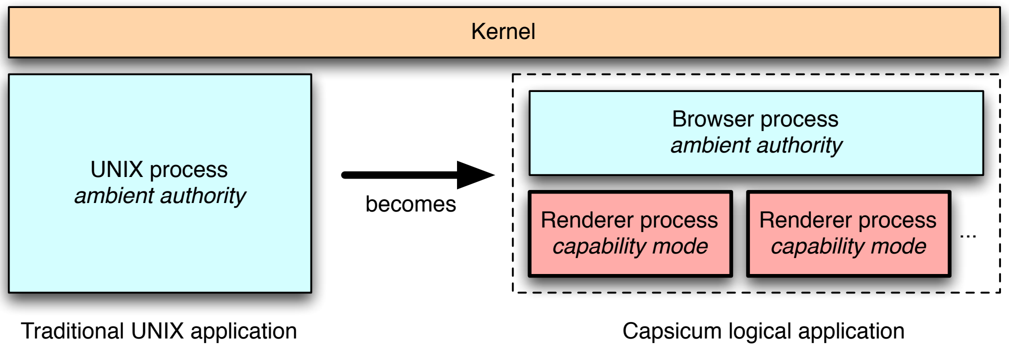

Figure 1: Capsicum helps applications self-compartmentalise.

Capsicum extends, rather than replaces, standard UNIX APIs by adding kernel-level primitives (a sandboxed *capability mode*, *capabilities* and others) and userspace support code (*libcapsicum* and a *capability-aware run-time linker*). Together, these extensions support application *compartmentalisation*, the decomposition of monolithic application code into components that will run in independent sandboxes to form *logical applications*, as shown in Figure 1.

Capsicum requires application modification to exploit new security functionality, but this may be done gradually, rather than requiring a wholesale conversion to a pure capability model. Developers can select the changes that maximize positive security impact while minimizing unacceptable performance costs; where Capsicum replaces existing sandbox technology, a performance improvement may even be seen.

This model required a number of pragmatic design choices, not least the decision to eschew micro-kernel architecture and migration to pure message-passing. While applications may adopt a message-passing approach, and indeed will need to do so to fully utilize the Capsicum architecture, we provide "fast paths" in the form of direct system call manipulation of kernel objects through delegated file descriptors. This allows native UNIX performance for file system I/O, network access, and other critical operations, while leaving the door open to techniques such as message-passing system calls for cases where that proves desirable.

2.1 Capability mode

Capability mode is a process credential flag set by a new system call, `cap_enter`; once set, the flag is inherited by all descendent processes, and cannot be cleared. Processes in capability mode are denied access to global namespaces such as the filesystem and PID namespaces (see Figure 2).[2]

Namespace	Description
Process ID (PID)	UNIX processes are identified by unique IDs. PIDs are returned by `fork` and used for signal delivery, debugging, monitoring, and status collection.
File paths	UNIX files exist in a global, hierarchical namespace, which is protected by discretionary and mandatory access control.
NFS file handles	The NFS client and server identify files and directories on the wire using a flat, global file handle namespace. They are also exposed to processes to support the lock manager daemon and optimise local file access.
File system ID	File system IDs supplement paths to mount points, and are used for forceable unmount when there is no valid path to the mount point.
Protocol addresses	Protocol families use socket addresses to name local and foreign endpoints. These exist in global namespaces, such as IPv4 addresses and ports, or the file system namespace for local domain sockets.
Sysctl MIB	The `sysctl` management interface uses numbered and named entries, used to get or set system information, such as process lists and tuning parameters.
System V IPC	System V IPC message queues, semaphores, and shared memory segments exist in a flat, global integer namespace.
POSIX IPC	POSIX defines similar semaphore, message queue, and shared memory APIs, with an undefined namespaces: on some systems, these are mapped into the file system; on others they are simply a flat global namespaces.
System clocks	UNIX systems provide multiple interfaces for querying and manipulating one or more system clocks or timers.
Jails	The management namespace for FreeBSD-based virtualised environments.
CPU sets	A global namespace for affinity policies assigned to processes and threads.

Figure 2: Global namespaces in the FreeBSD operating kernel

Access to system calls in capability mode is also restricted: some system calls are unavailable (those which require global namespace access), while others are constrained. For instance, `sysctl` can be used to

[2]In addition to these namespaces, there are several system management interfaces that must be protected to maintain UNIX process isolation. These interfaces include `/dev` device nodes that allow physical memory or PCI bus access, some `ioctl` operations on sockets, and management interfaces such as `reboot` and `kldload`, which loads kernel modules.

query process-local information such as address space layout, but also to monitor a system's network connections. We have constrained `sysctl` by explicitly marking ≈30 of 3000 that do not violate containment.

The system calls which require constraints are `sysctl`, `shm_open`, which is permitted to create *anonymous memory objects*, but not named ones, and the `openat` family of system calls. These calls already accept a file descriptor argument as the directory to perform the open, rename, etc. relative to; in capability mode, they are constrained so that they can only operate on objects "under" this descriptor[3].

2.2 Capabilities

The most critical decision in adding capability support to a UNIX system is the relationship between capabilities and file descriptors. Some systems, such as Mach/BSD, have maintained entirely independent notions: Mac OS X provides each task with both indexed capabilities (ports) and file descriptors. Separating these concerns is logical, as Mach ports have different semantics from file descriptors; however, confusing results can arise for application developers dealing with both Mach and BSD APIs, and we wanted to reuse existing APIs as much as possible. As a result, we chose to extend the file descriptor abstraction, and introduce a new file descriptor type, the capability, to wrap and protect raw UNIX file descriptors.

File descriptors already have some properties of capabilities: they are unforgeable tokens of authority, and can be inherited by a child process or passed between processes that share an IPC channel. Unlike "pure" capabilities, however, they confer very broad rights: even if a file descriptor is read-only, operations on meta-data such as `fchmod` are permitted. In the Capsicum model, we restrict these operations by wrapping the descriptor in a capability and permitting only authorized operations via the capability, as shown in Figure 3.

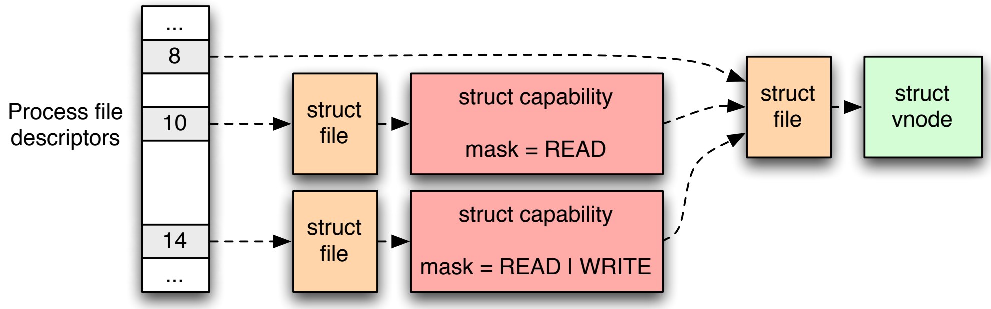

Figure 3: Capabilities "wrap" normal file descriptors, masking the set of permitted methods.

The `cap_new` system call creates a new capability given an existing file descriptor and a mask of rights; if the original descriptor is a capability, the requested rights must be a subset of the original rights. A capability's rights are checked by the kernel code converting a file descriptor number to an in-kernel reference, giving us confidence that no paths exist to access file descriptors without capability checks.

There are roughly 60 capability rights, striking a balance between message-passing (two rights: send and receive), and MAC systems (hundreds of access control checks). We assigned rights based on logical methods: system calls implementing semantically identical operations require the same rights, and some calls may require multiple rights. For example, `pread` (read to memory) and `preadv` (read to a memory vector) both require `CAP_READ`, and `read` (read bytes using the file offset) requires `CAP_READ | CAP_SEEK`.

Capabilities can wrap any type of file descriptor including directories, which can then be passed as arguments to `openat` and related system calls. The `*at` system calls begin relative lookups for file operations

[3]For instance, if file descriptor 4 is a capability allowing access to /lib, then `openat(4, "libc.so.7")` will succeed, whereas `openat(4, "../etc/passwd")` and `openat(4, "/etc/passwd")` will not.

with the directory descriptor; we disallow some cases when a capability is passed: absolute paths, paths containing ".." components, and `AT_CWD`, which requests a lookup relative to the current working directory. With these constraints, directory capabilities delegate file system namespace subsets, as shown in Figure 4. This allows sandboxed processes to access multiple files in a directory (such as the library path) without the performance overhead or complexity of proxying each file `open` via IPC to a process with ambient authority.

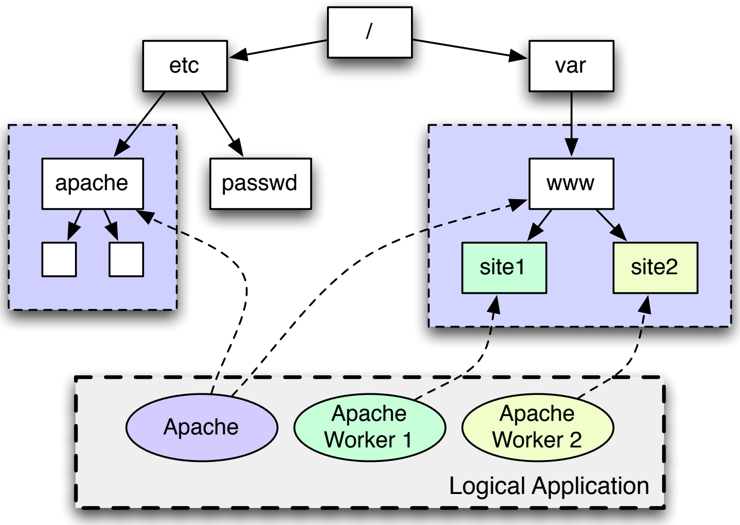

Figure 4: Portions of the global filesystem namespace can be delegated to sandboxed processes.

The ".." restriction is a conservative design, and prevents a subtle problem similar to historic `chroot` vulnerabilities. A single directory capability that only enforces containment by preventing ".." lookup on the root of a delegated subtree operates correctly; however, two colluding sandboxes (or a single sandbox with two capabilities) can race to actively arrange a tree so that this check always passes, allowing escape from the delegated subset. It is possible to imagine less conservative solutions, such as preventing upward renames that could introduce exploitable cycles during lookup, or additional synchronisation; these strike us as more risky tactics, and we have selected the simplest solution, at some cost to flexibility.

2.3 Run-time environment

Even with Capsicum's kernel primitives, creating sandboxes without leaking undesired resources via file descriptors, memory mappings, or memory contents is difficult. `libcapsicum` therefore provides an API for starting scrubbed sandbox processes, and explicit delegation APIs to assign rights to sandboxes. `libcapsicum` cuts off the sandbox process' access to global namespaces via `cap_enter`, but also closes file descriptors not positively identified for delegation to the sandbox and flushes the address space. Sandbox creation returns a UNIX domain socket for applications to use for inter-process communication (IPC) between host and sandbox; this socket can be used to grant additional rights to the sandbox as required.

3 Capsicum implementation

3.1 Kernel changes

Many system call and capability constraints are applied at the point of implementation of kernel services, rather than by simply filtering system calls. The advantage of this approach is that a single constraint, such as the blocking of access to the global file system namespace, can be implemented in one place, `namei`, which is responsible for processing all path lookups. For example, one might not have expected the `fexecve` call to cause global namespace access, since it takes a file descriptor as its argument rather than a path for the binary to execute. However, the file passed by file descriptor specifies its run-time linker via a path embedded in the binary, which the kernel will then open and execute.

Similarly, capability rights are checked by the kernel function `fget`, which converts a numeric descriptor into a `struct file` reference. We have added a new `rights` argument, allowing callers to declare what capability rights are required to perform the current operation. If the file descriptor is a raw UNIX descriptor, or wrapped by a capability with sufficient rights, the operation succeeds. Otherwise, `ENOTCAPABLE` is returned. Changing the signature of `fget` allows us to use the compiler to detect missed code paths, providing greater assurance that all cases have been handled.

One less trivial global namespace to handle is the process ID (PID) namespace, which is used for process creation, signaling, debugging and exit status, critical operations for a logical application. Another problem for logical applications is that libraries cannot create and manage worker processes without interfering with process management in the application itself—unexpected `SIGCHLD` signals are delivered to the application, and unexpected process IDs are returned by `wait`.

Process descriptors address these problems in a manner similar to Mach task ports: creating a process with `pdfork` returns a file descriptor that can be used for process management tasks, such as monitoring for exit via `poll`. `pdfork` suppresses generation of `SIGCHLD` on process termination, and prevents its PID being returned by `wait`. This leads to a user experience consistent with that of monolithic processes: when a user hits Ctrl-C, or the application segfaults, all sub-components of the logical application terminate[4].

3.2 The Capsicum run-time environment

Removing access to global namespaces forces fundamental changes to the UNIX run-time environment. Even the most basic UNIX operations for starting processes and running programs have been eliminated: `fork` and `exec` both rely on global namespaces. Responsibility for launching a sandbox is shared. `libcapsicum` is invoked by the application, and responsible for forking a new process, gathering together delegated capabilities from both the application and run-time linker, and directly executing the run-time linker, passing the sandbox binary via a capability. By directly running the run-time linker, we avoid the need for the kernel to use a hard-code path to the run-time linker in the ELF header, which is not permitted in capability mode.

Once `rtld-elf-cap` is executing in the new process, it loads and links the sandbox binary using shared libraries loaded from library directory capabilities set up by `libcapsicum`. Once the sandbox is in execution, started via the `cap_main` entry point rather than the traditional `main` entry point, libcapsicum will respond to application requests for delegated capabilities. Using a separate entry point makes it easier to use a single binary to hold both unsandboxed code and code intended to run in capability mode, rather than each application establishing its own conventions for command line arguments, etc, to specify these two, typically quite different, code paths. This process is illustrated in greater detail in Figure 5.

Once in execution, the application is linked against normal C libraries and has access to much of the traditional C run-time, subject to the availability of system calls that the run-time depends on. An IPC

[4]This termination will currently not occur if reference cycles exist among processes, suggesting the need for a "logical application" primitive—see Section 7.

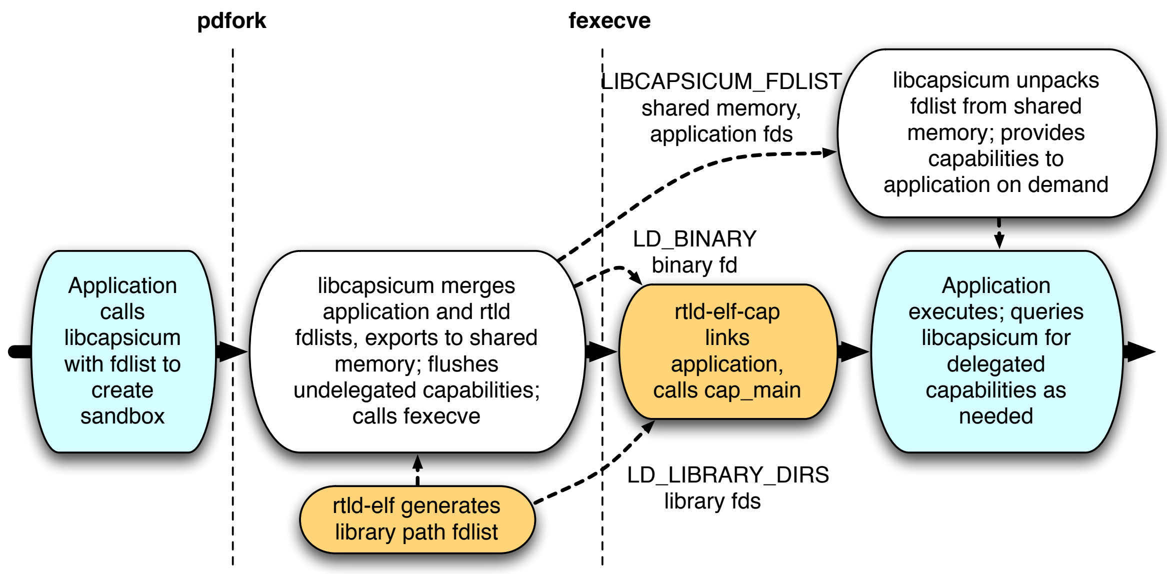
pdfork
fexecve
Application calls libcapsicum with fdlist to create sandbox
libcapsicum merges application and rtld fdlists, exports to shared memory; flushes undelegated capabilities; calls fexecve
rtld-elf generates library path fdlist
LIBCAPSICUM_FDLIST shared memory, application fds
libcapsicum unpacks fdlist from shared memory; provides capabilities to application on demand
LD_BINARY binary fd
rtld-elf-cap links application, calls cap_main
LD_LIBRARY_DIRS library fds
Application executes; queries libcapsicum for delegated capabilities as needed

Figure 5: Process and components involved in creating a new `libcapsicum` sandbox

channel, in the form of a UNIX domain socket, is set up automatically by `libcapsicum` to carry RPCs and capabilities delegated after the sandbox starts. Capsicum does not provide or enforce the use of a specific Interface Description Language (IDL), as existing compartmentalized or privilege-separated applications have their own, often hand-coded, RPC marshaling already. Here, our design choice differs from historic capability systems, which universally have selected a specific IDL, such as the Mach Interface Generator (MIG) on Mach.

`libcapsicum`'s fdlist abstraction allows complex, layered applications to declare capabilities to be passed into sandboxes, in effect providing a sandbox template mechanism. This avoids encoding specific file descriptor numbers into the ABI between applications and their sandbox components, a technique we found used in Chromium that is likely to lead to programmer error. Of particular concern is hard-coding of file descriptor numbers for specific purposes, when those descriptor numbers may already have been used by other layers of the system. Instead, application and library components declare process-local names bound to file descriptor numbers before creating the sandbox, and then matched components in the sandbox query those names to retrieve (possibly renumbered) file descriptors.

4 Adapting applications to use Capsicum

Adapting applications for use with sandboxing is a non-trivial task, regardless of the framework, as it requires analyzing programs to determine their resource dependencies, and adopting a distributed system programming style in which components must use message passing or explicit shared memory rather than relying on a common address space for communication. In Capsicum, programmers have a choice of working directly with capability mode or using `libcapsicum` to create and manage sandboxes, and each model has its merits and costs in terms of development complexity, performance impact, and security:

1. Modify applications to use `cap_enter` directly in order to convert an existing process with ambient privilege into a capability mode process inheriting only specific capabilities via file descriptors and virtual memory mappings. This works well for applications with a simple structure like: open all

resources, then process them in an I/O loop, such as programs operating in a UNIX pipeline, or interacting with the network for the purposes of a single connection. The performance overhead will typically be extremely low, as changes consist of encapsulating broad file descriptor rights into capabilities, followed by entering capability mode. We illustrate this approach with `tcpdump`.

2. Use `cap_enter` to reinforce the sandboxes of applications with existing privilege separation or compartmentalisation. These applications have a more complex structure, but are already aware that some access limitations are in place, so have already been designed with file descriptor passing in mind. Refining these sandboxes can significantly improve security in the event of a vulnerability, as we show for `dhclient` and Chromium; the performance and complexity impact of these changes will be low because the application already adopts a message passing approach.

3. Modify the application to use the full `libcapsicum` API, introducing new compartmentalisation in the application, or reformulating existing privilege separation. This offers significantly stronger protection, by virtue of flushing capability lists and residual memory from the host environment, but at higher development and run-time costs. Boundaries must be identified in the application such that not only is security improved (i.e., code processing risky data is isolated), but so that resulting performance is sufficiently efficient. We illustrate this technique using modifications to `gzip`.

Compartmentalized application development is, of necessity, distributed application development, with software components running in different processes and communicating via message passing. Distributed debugging is an active area of research, but commodity tools are unsatisfying and difficult to use. While we have not attempted to extend debuggers, such as `gdb`, to better support distributed debugging, we have modified a number of FreeBSD tools to improve support for capability mode development, and take some comfort in the generally synchronous nature of most compartmentalized applications.

The FreeBSD `procstat` command inspects kernel-related state of running processes, including file descriptors, virtual memory mappings, and security credentials. In Capsicum, these resource lists become capability lists, representing the only rights available to the process. We have extended these modes to show new Capsicum-related information, such the as capability rights masks on file descriptors and a flag in process credential listings to indicate capability mode. As a result, developers can directly inspect the capabilities inherited or passed to sandboxes.

When adapting existing software to run in capability mode, identifying capability requirements can be tricky; often the best technique is to discover these through dynamic analysis, identifying missing dependencies by tracing real-world use. To this end, capability-related failures return a new `errno` value, `ENOTCAPABLE` distinguishing them from other security failures, and system calls such as `open` are permitted to enter the kernel before being blocked in `namei` so that paths are submitted to FreeBSD's `ktrace` facility, or utilized in `DTrace` scripts.

Another common compartmentalised development strategy is to allow the multi-process logical application to be run as a single process for debugging purposes. `libcapsicum` provides an API to query whether sandboxing for the current application or component is enabled by policy, making it easy to enable and disable sandboxing for testing. As RPCs are generally synchronous, the thread stack in the sandbox process is logically an extension of the thread stack in the host process, which makes the distributed debugging task less fraught than it otherwise might appear.

4.1 tcpdump

`tcpdump` provides an excellent example of Capsicum primitives offering immediate wins through straightforward changes, but also the subtleties that arise when compartmentalising software not written with that goal in mind. `tcpdump` has a simple model: compile a pattern into a BPF filter, configure a BPF device as an input source, and loop writing captured packets rendered as text. This structure lends itself to sandboxing: resources are acquired early with ambient privilege, and later processing depends only on held capabilities, so can execute in capability mode. A two-line change implements this conversion:

```
+       if (cap_enter() < 0)
+               error("cap_enter: %s", pcap_strerror(errno));
        status = pcap_loop(pd, cnt, callback, pcap_userdata);
```

This significantly improves security, as historically fragile packet-parsing code now executes with reduced privilege. However, further analysis is required to confirm that only desired capabilities are held exposed:

```
% procstat -fC 19184
  PID COMM              FD T     FLAGS CAPABILITIES PRO NAME
19184 tcpdump          cwd v ---------            - -   /usr/home/robert
19184 tcpdump         root v ---------            - -   /
19184 tcpdump            0 v rw-------            - -   /dev/pts/2
19184 tcpdump            1 v rw-------            - -   /dev/pts/2
19184 tcpdump            2 v rw-------            - -   /dev/pts/2
19184 tcpdump            3 v rw-------            - -   /dev/bpf
```

While there are few surprises, unconstrained access to a user's terminal connotes significant rights, such as reading key presses. This refinement prevents reading `stdin`, while still allowing normal output:

```
+       if (lc_limitfd(STDIN_FILENO, CAP_FSTAT) < 0)
+               error("lc_limitfd: unable to limit STDIN_FILENO");
+       if (lc_limitfd(STDOUT_FILENO, CAP_FSTAT | CAP_SEEK | CAP_WRITE) < 0)
+               error("lc_limitfd: unable to limit STDOUT_FILENO");
+       if (lc_limitfd(STDERR_FILENO, CAP_FSTAT | CAP_SEEK | CAP_WRITE) < 0)
+               error("lc_limitfd: unable to limit STDERR_FILENO");
```

`ktrace` reveals another problem, `libc` DNS resolver code depends on file system access, but not until after `cap_enter`, leading to denied access and lost functionality:

```
 19318 tcpdump  CALL  open(0x800caebc3,O_RDONLY,<unused>0x1b6)
 19318 tcpdump  NAMI  "/etc/resolv.conf"
 19318 tcpdump  RET   open -1 errno 45 Operation not supported
```

This illustrates a subtle problem with sandboxing: highly layered software designs often rely on on-demand initialisation, lowering or avoiding startup costs, and those initialisation points are scattered across many components in system and application code. This is corrected by switching to the lightweight resolver, which sends DNS queries to a local daemon that performs actual resolution, addressing both file system and network address namespace concerns. Despite these limitations, this example of capability mode and capability APIs shows that even minor code changes can lead to dramatic security improvements, especially for a critical application with a long history of security problems.

4.2 dhclient

FreeBSD ships the OpenBSD DHCP client, which includes privilege separation support. On BSD systems, the DHCP client must run with privilege to open BPF descriptors, create raw sockets, and configure network interfaces. This creates an appealing target for attackers: network code exposed to a complex packet format while running with root privilege. The DHCP client is afforded only weak tools to constrain operation: it starts as the root user, opens the resources its unprivileged component will require (raw socket, BPF descriptor, lease configuration file), forks a process to continue privileged activities (such as network configuration), and then confines the parent process using chroot and the setuid family of system calls. Despite hardening of the BPF ioctl interface to prevent reattachment to another interface or reprogramming the filter, this confinement is weak; chroot limits only file system access, and switching credentials offers poor protection against weak or incorrectly configured DAC protections on the sysctl and PID namespaces.

Through a similar two-line change to that in tcpdump, we can reinforce (or, through a larger change, replace) existing sandboxing with capability mode. This instantly denies access to the previously-exposed global namespaces, while permitting continued use of held file descriptors. As there has been no explicit flush of address space, memory, or file descriptors, it is important to analyze what capabilities have been leaked into the sandbox, the key limitation to this approach. We use procstat -fC to analyze the file descriptor array:

```
  PID COMM                FD T      FLAGS CAPABILITIES PRO NAME
18988 dhclient           cwd v ---------            - -   /var/empty
18988 dhclient          root v ---------            - -   /var/empty
18988 dhclient          jail v ---------            - -   /var/empty
18988 dhclient             0 v rw-------            - -   /dev/null
18988 dhclient             1 v rw-------            - -   /dev/null
18988 dhclient             2 v rw-------            - -   /dev/null
18988 dhclient             3 s rw-------            - UDD /var/run/logpriv
18988 dhclient             5 s rw-------            - ?
18988 dhclient             6 p rw-------            - -   -
18988 dhclient             7 v -w-------            - -   /var/db/dhclient.leas
18988 dhclient             8 v rw-------            - -   /dev/bpf
18988 dhclient             9 s rw-------            - IP? 0.0.0.0:0 0.0.0.0:0
```

The existing dhclient code has done an effective job at eliminating directory access, but continues to allow the sandbox direct rights to submit arbitrary log messages to syslogd, modify the lease database, and a raw socket on which a broad variety of operations could be performed. The last of these is of particular interest due to ioctl; although dhclient has given up system privilege, many network socket ioctls are defined, allowing access to system information. These are blocked in Capsicum's capability mode.

It is easy to imagine extending existing privilege separation in dhclient to use the Capsicum capability facility to further constrain file descriptors inherited in the sandbox environment, for example, by limiting use of the IP raw socket to send and recv, disallowing ioctl. Use of the libcapsicum API would require more significant code changes, but as dhclient already adopts a message passing structure to communicate with its components, it would be relatively straight forward, offering better protection against capability and memory leakage. Further migration to message passing would prevent arbitrary log messages or direct unformatted writes to dhclient.leases.em by constraining syntax.

4.3 gzip

The gzip command line tool presents an interesting target for conversion for several reasons: it implements risky compression/decompression routines that have suffered past vulnerabilities, it contains no existing compartmentalisation, and it executes with ambient user (rather than system) privileges. Historic UNIX sandboxing techniques, such as chroot and ephemeral UIDs are a poor match because of their privilege requirement, but also because (unlike with dhclient), there's no expectation that a single sandbox exist—many gzip sessions can run independently for many different users, and there can be no assumption that placing them in the same sandbox provides the desired security properties.

The first step is to identify natural fault lines in the application: for example, code that requires ambient privilege (due to opening files or building network connections), and code that performs more risky activities, such as parsing data and managing buffers. In `gzip`, this split becomes immediately obvious: the main run loop of the application processes command line arguments, identifies streams and objects to perform processing on and send results to, and then feeds them to compress routines that accept input and output file descriptors. This suggests a natural partitioning in which pairs of descriptors are submitted to a sandbox for processing after the ambient privilege process opens them and performs initial header handling.

We modified `gzip` to use `libcapsicum`, intercepting three core functions and optionally proxying them to a sandbox based on policy queried from `libcapsicum` using three RPCs:

Function	RPC	Description
`gz_compress`	`PROXIED_GZ_COMPRESS`	zlib-based compression
`gz_uncompress`	`PROXIED_GZ_UNCOMPRESS`	zlib-based decompression
`unbzip2`	`PROXIED_UNBZIP2`	bzip2-based decompression

Each RPC passes two file descriptors, one for input and one for output, to the sandbox, as well as miscellaneous fields such as returned size, original filename, and modification time. By limiting capability rights on the passed descriptors to a combination of `CAP_READ`, `CAP_WRITE`, and `CAP_SEEK`, a tightly constrained sandbox is created, preventing access to any other files in the file system, or other globally named resources, in the event a vulnerability in compression code is exploited.

These changes add 409 lines (about 16%) to the size of the `gzip` source code, largely to marshal and un-marshal RPCs. In adapting `gzip`, we were initially surprised to see a performance improvement when sandboxed; this unlikely result revealed that we had failed to propagate the compression level argument into the sandbox, leading to the incorrect algorithm selection. This oversight occurred because this argument was passed via a global variable rather than an explicit function arguments. This serves as reminder that code not originally written for decomposition requires careful analysis, and oversights such as this one will not be caught by the compiler: the variable was correctly defined in both processes, but its value never propagated.

4.4 Chromium

Google's Chromium web browser uses a multi-process architecture similar to a Capsicum logical application to improve robustness [18]. In this model, each tab is associated with a *renderer process* that performs the risky and complex task of rendering page contents through page parsing, image rendering, and JavaScript execution. More recent work on Chromium has integrated sandboxing techniques to improve resilience to malicious attacks rather than occasional instability; this has been done in various ways on different supported operating systems, as we will discuss in detail in Section 5.

The FreeBSD port of Chromium did not include sandboxing, and the sandboxing facilities provided as part of the similar Linux and Mac OS X ports bear little resemblance to Capsicum. However, the existing compartmentalisation meant that several critical tasks had already been performed:

- Chromium assumes that processes can be converted into sandboxes that limit new object access
- Certain services were already forwarded to renderers, such as font loading
- Shared memory is used to transfer output between renderers and the web browser
- Chromium contains RPC marshaling and passing code in all the required places

The only significant Capsicum change to the FreeBSD port of Chromium was to switch from System V shared memory (permitted in Linux sandboxes) to the POSIX shared memory code used in the Mac OS X port (capability-oriented and permitted in Capsicum's capability mode). Approximately 100 additional lines of code were required to introduce calls to `lc_limitfd` to limit access to file descriptors inherited by sandbox processes, such as Chromium data `pak` files, `stdio`, and `/dev/random`, and to call `cap_enter`. This compares favourably with the 4.3 million lines of code in the Chromium source tree, but would not have been possible without existing sandbox support in the design. We believe it should be possible, without a significantly higher number of lines of code, to explore using the `libcapsicum` API directly.

5 A comparison of sandboxing technologies

We now compare Capsicum's compartmentalisation model to existing sandbox mechanisms. Chromium provides an ideal context for this comparison, as it takes advantage of six sandboxing technologies (see Figure 6). Of these, the two are DAC-based, two MAC-based and two capability-based.

Operating system	Model	Line count	Description
Windows	ACLs	22,350	Windows ACLs and SIDs
Linux	`chroot`	605	`setuid` root helper sandboxes renderer
Mac OS X	Seatbelt	560	Path-based MAC sandbox
Linux	SELinux	200	Restricted sandbox type enforcement domain
Linux	`seccomp`	11,301	`seccomp` and userspace syscall wrapper
FreeBSD	Capsicum	100	Capsicum sandboxing using `cap_enter`

Figure 6: Sandboxing mechanisms employed by Chromium.

5.1 Windows ACLs and SIDs

On Windows, Chromium uses a DAC-based mechanism to create sandboxes [18]. The unsuitability of inter-user protections for the intra-user context is demonstrated well: the model is both incomplete and unwieldy. Chromium uses Access Control Lists (ACLs) and Security Identifiers (SIDs) to sandbox renderers on Windows. Chromium creates a modified, reduced privilege, SID, which does not appear in the ACL of any object in the system, in effect running the renderer as an anonymous "nobody" user[5].

However, objects which do not support ACLs do not restrict access by sandboxes. Additional precautions such as an alternate, invisible desktop isolate sandboxes from the user's GUI environment, but other ACL-free objects are unprotected. Such objects include FAT filesystems on USB sticks and TCP/IP sockets: a sandbox cannot read user files directly, but it may be able to communicate with any server on the Internet! [6]

Many legitimate system calls are also denied to the sandboxed process. These system calls must be forwarded by the sandbox module to a trusted process, which is responsible for filtering and serving them. This forwarding comprises most of the 22,000 lines of code in the Windows sandbox module.

5.2 Linux chroot

Chromium's `suid` sandbox mechanism on Linux also attempts to create a privilege-free sandbox using historic OS access control; however, the result is similarly porous, with the added risk that it requires system privilege to create the sandbox.

In this model, access to the filesystem is limited to a directory via `chroot`: the directory becomes the sandbox's virtual root directory. Access to other namespaces, including System V shared memory (where

[5]The Chromium Windows sandbox documentation is quick to point out that this is not strictly the case: the nature of the SID is such that auditable events are still attributed to the authenticated user.

[6]USB sticks present a significant concern, as they are frequently used for file sharing, backup, and protection from malware.

the user's X window server can be contacted) and network access, is unconstrained, and great care must be taken to avoid leaking resources when entering the sandbox.

Furthermore, initiating `chroot` requires a `setuid` binary: a program that runs with full system privilege. While comparable to Capsicum's capability mode in terms of intent, this model suffers significant sandboxing weakness (for example, permitting full access to the System V shared memory as well as all operations on passed file descriptors), and comes at the cost of an additional setuid-root binary that runs with system privilege.

5.3 MAC OS X Seatbelt

On Mac OS X, Chromium uses a MAC-based framework for creating sandboxes. This allows Chromium to create a stronger sandbox than is possible with DAC, but the rights that are granted to render processes are still very broad, and security policy must be specified separately from the code that relies on it.

The Mac OS X *Seatbelt* sandboxing system allows processes to be constrained according to a LISP-based policy language [1]. It uses MAC Framework to check application activities against this policy; Chromium uses three policies for different components, allowing access to filesystem elements such as font directories while restricting access to the global namespace.

Like other techniques, resources are acquired before constraints are imposed, so care must be taken to avoid leaking resources into the sandbox. Fine-grained filesystem constraints are possible, but other namespaces such as the POSIX shared memory namespace, are an all-or-nothing affair.

The Seatbelt-based sandbox model is less verbose than other approaches, but like all MAC systems, security policy must be expressed separately from code. This can lead to inconsistencies and vulnerabilities.

5.4 SELinux

Chromium's MAC approach on Linux uses SELinux via a Type Enforcement policy [12]. SELinux can be used for very fine-grained sandboxing, but in practice, broad rights are conferred because fine-grained Type Enforcement policies are difficult to write and maintain. The requirement that an administrator be involved in defining new policy and applying new types to the file system is a significant inflexibility: application policies cannot adapt dynamically, as system privilege is required to reformulate policy and relabel objects.

The Fedora reference policy for Chromium creates a SELinux dynamic domain, `chrome_sandbox_t`, which is shared by all sandboxes, risking potential interference between sandboxes. This domain is assigned broad rights, such as the ability to read all files in `/etc` and access to the terminal device. These broad policies are easier to craft than fine-grained ones, reducing the impact of the dual-coding problem, but they are also less effective.

5.5 Linux seccomp

Linux provides an optionally-compiled capability mode-like facility called `seccomp`. Processes in `seccomp` mode are denied access to all system calls except `read`, `write`, and `exit`. At face value, this seems promising, but as OS infrastructure to support applications using `seccomp` is minimal, application writers must go to significant effort to use it.

In order to allow other system calls, Chromium constructs a process in which one thread executes in `seccomp` mode, and another "trusted" thread that shares the same address space but has normal system call access. Chromium rewrites `glibc` and other library system call vectors to pass system calls via shared memory to the trusted thread, where they are filtered in order to prevent access to inappropriate shared memory objects, opening files for write, etc. However, this default policy is, itself, quite weak, as read of any file system object is permitted.

The Chromium `seccomp` sandbox includes over a thousand lines of hand-crafted assembly code to set up the sandbox, implement system call forwarding, a trusted thread, and a basic security policy. Such code is a risky proposition: difficult to write and maintain, with bugs likely leading to lead to security vulnerabilities.

6 Performance evaluation

Typical operating system security benchmarking is targeted at illustrating zero or near-zero overhead in the hopes of selling general applicability of the resulting technology. Our thrust is slightly different: we know that application authors who have already begun to adopt compartmentalization are willing to accept significant overheads for mixed security return. Our goal is therefore to accomplish comparable performance with significantly improved security.

We evaluate performance in two ways: first, a set of micro-benchmarks establishing the overhead introduced by Capsicum's capability mode and capability primitives. As we are unable to measure any noticeable performance change in our adapted UNIX applications (`tcpdump` and `dhclient`) due to the extremely low cost of entering capability mode from an existing process, we then turn our attention to the performance of our `libcapsicum`-enhanced `gzip`.

All performance measurements have been performed on an 8-core Intel Xeon E5320 system running at 1.86GHz with 4GB of RAM, running either an unmodified FreeBSD 8-STABLE distribution synchronized to revision 201781 (2010-01-08) from the FreeBSD Subversion repository, or a synchronised 8-STABLE distribution with our capability enhancements.

6.1 System call performance

First, we consider system call performance through micro-benchmarking. Figure 7 summarizes these results for various system calls on unmodified FreeBSD, and related capability operations in Capsicum. Figure 6.2 contains a table of benchmark timings.

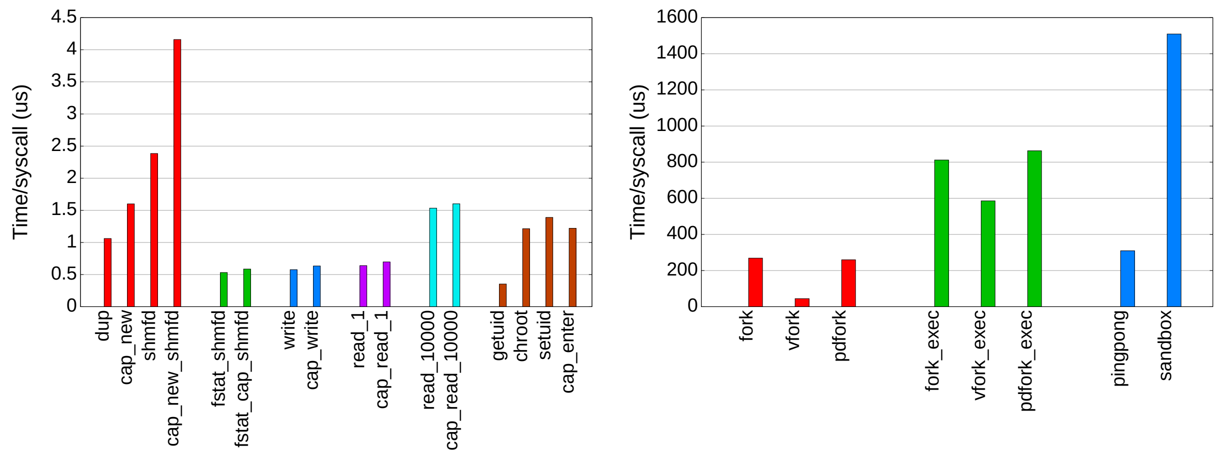

Figure 7: Capsicum system call performance compared to standard UNIX calls.

Our first concern is with the performance of capability creation, as compared to raw object creation and the closest UNIX operation, `dup`. We observe moderate, but expected, performance overheads for capability wrapping of existing file descriptors: the `cap_new` syscall is $50.7\% \pm 0.08\%$ slower than dup, or 539 ± 0.8ns slower in absolute terms[7].

Next, we consider the overhead of capability "unwrapping", which occurs on every descriptor operation. We compare the cost of some simple operations on raw file descriptors, to the same operations on a capability-wrapped version of the same file descriptor: writing a single byte to `/dev/null`, reading a single byte from `/dev/zero`; reading 10000 bytes from `/dev/zero`; and performing a `fstat` call on a shared memory file descriptor. In all cases we observe a small overhead of about $0.06\mu s$ when operating on the capability-wrapped file descriptor. This has the largest relative performance impact on `fstat` (since it does not perform I/O, simply inspecting descriptor state, it should thus experience the highest overhead of any system call which requires unwrapping). Even in this case the overhead is relatively low: $10.2\% \pm 0.5\%$.

6.2 Sandbox creation

Capsicum supports ways to create a sandbox: directly invoking `cap_enter` to convert an existing process into a sandbox, inheriting all current capability lists and memory contents, and the `libcapsicum` sandbox API, which creates a new process with a flushed capability list.

`cap_enter` performs similarly to `chroot`, used by many existing compartmentalized applications to restrict file system access. However, `cap_enter` out-performs `setuid` as it does not need to modify resource limits. As most sandboxes `chroot` and set the UID, entering a capability mode sandbox is roughly twice as fast as a traditional UNIX sandbox. This suggests that the overhead of adding capability mode support to an application with existing compartmentalisation will negligible, and replacing existing sandboxing with `cap_enter` may even marginally improve performance.

Creating a new sandbox process and replacing its address space using `execve`, however, is an expensive operation. Micro-benchmarks indicate that the cost of `fork` is three orders of magnitude more expensive than manipulating the process credential, and further adding `execve` or even a single instance of message passing increases that cost further[8]. Creating, exchanging an RPC with, and destroying a single sandbox (the "sandbox" label in Figure 7(b)) has a cost of about 1.5ms, significantly higher than its subset components.

6.3 gzip performance

While the performance cost of `cap_enter` is negligible compared to other activity, the cost of multi-process sandbox creation (already taken by `dhclient` and Chromium due to existing sandboxing) is significant.

To measure the cost of process sandbox creation, we timed `gzip` compressing files of various sizes. Since the additional overheads of sandbox creation are purely at startup, we expect to see a constant-time overhead to the capability-enhanced version of `gzip`, with identical linear scaling of compression performance with input file size. Files were pre-generated on a memory disk by reading from a constant-entropy data source[9] (the use of a data source with approximately constant entropy per bit minimizes variation in overall `gzip` performance due to changes in the compressor performance as files of different sizes are sampled). The list of files was piped to `xargs -n 1 gzip -c > /dev/null`, which sequentially invokes a new `gzip` compression process with a single file argument, and discards the compressed output. Sufficiently many input files were generated to provide at least 10 seconds of repeated `gzip` invocations, and the overall run-time measured. Since the files were staged on a memory disk, I/O overhead was minimized. The use of

[7]The micro-benchmarks were run by performing the target operation in a tight loop over an interval of at least 10 seconds, repeating for 10 iterations. Differences were computed using Student's t-test at 95% confidence.

[8]We also found that additional dynamically linked library dependencies (`libcapsicum` and its system library dependency `libsbuf`) imposed an additional 9% cost to the `fork` syscall, presumably due to the additional virtual memory mappings being copied to the child process. This overhead was not present on `vfork` which we plan to use in `libcapsicum` in the future.

[9]`/dev/zero` for perfectly compressible data, `/dev/random` for perfectly incompressible data, and base 64-encoded `/dev/random` for a moderate high entropy data source, with about 24% compression after gzipping.

Benchmark	Time/operation	Difference	% difference
`dup`	$1.061 \pm 0.000\mu s$	-	-
`cap_new`	$1.600 \pm 0.001\mu s$	$0.539 \pm 0.001\mu s$	$50.7\% \pm 0.08\%$
`shmfd`	$2.385 \pm 0.000\mu s$	-	-
`cap_new_shmfd`	$4.159 \pm 0.007\mu s$	$1.77 \pm 0.004\mu s$	$74.4\% \pm 0.181\%$
`fstat_shmfd`	$0.532 \pm 0.001\mu s$	-	-
`fstat_cap_shmfd`	$0.586 \pm 0.004\mu s$	$0.054 \pm 0.003\mu s$	$10.2\% \pm 0.506\%$
`read_1`	$0.640 \pm 0.000\mu s$	-	-
`cap_read_1`	$0.697 \pm 0.001\mu s$	$0.057 \pm 0.001\mu s$	$8.93\% \pm 0.143\%$
`read_10000`	$1.534 \pm 0.000\mu s$	-	-
`cap_read_10000`	$1.601 \pm 0.003\mu s$	$0.067 \pm 0.002\mu s$	$4.40\% \pm 0.139\%$
`write`	$0.576 \pm 0.000\mu s$	-	-
`cap_write`	$0.634 \pm 0.002\mu s$	$0.058 \pm 0.001\mu s$	$10.0\% \pm 0.241\%$
`cap_enter`	$1.220 \pm 0.000\mu s$	-	-
`getuid`	$0.353 \pm 0.001\mu s$	$-0.867 \pm 0.001\mu s$	$-71.0\% \pm 0.067\%$
`chroot`	$1.214 \pm 0.000\mu s$	$-0.006 \pm 0.000\mu s$	$-0.458\% \pm 0.023\%$
`setuid`	$1.390 \pm 0.001\mu s$	$0.170 \pm 0.001\mu s$	$14.0\% \pm 0.054\%$
`fork`	$268.934 \pm 0.319\mu s$	-	-
`vfork`	$44.548 \pm 0.067\mu s$	$-224.3 \pm 0.217\mu s$	$-83.4\% \pm 0.081\%$
`pdfork`	$259.359 \pm 0.118\mu s$	$-9.58 \pm 0.324\mu s$	$-3.56\% \pm 0.120\%$
`pingpong`	$309.387 \pm 1.588\mu s$	$40.5 \pm 1.08\mu s$	$15.0\% \pm 0.400\%$
`fork_exec`	$811.993 \pm 2.849\mu s$	-	-
`vfork_exec`	$585.830 \pm 1.635\mu s$	$-226.2 \pm 2.183\mu s$	$-27.9\% \pm 0.269\%$
`pdfork_exec`	$862.823 \pm 0.554\mu s$	$50.8 \pm 2.83\mu s$	$6.26\% \pm 0.348\%$
`sandbox`	$1509.258 \pm 3.016\mu s$	$697.3 \pm 2.78\mu s$	$85.9\% \pm 0.339\%$

Figure 8: Micro-benchmark results for various system calls and functions, grouped by category.

`xargs` to repeatedly invoke `gzip` provides a tight loop that minimizes the time between `xargs`' successive `vfork` and `exec` calls of `gzip` itself. Each measurement was repeated 5 times and averaged.

Benchmarking `gzip` shows high initial overhead, when used to compress single-byte files, but also that the approach in which file descriptors are wrapped in capabilities and delegated rather than using pure message passing, leads to asymptotically identical behavior as file size increases and run-time cost are dominated by compression workload, which is unaffected by Capsicum. Specifically, we find that the overhead of launching a sandboxed `gzip` is 2.37 ± 0.01 ms, independent of the type of compression stream. For many workloads, this one-off performance cost is negligible, or can be amortized by passing multiple files to the same `gzip` invocation.

7 Future work

Capsicum provides an effective platform for capability work on UNIX platforms. However, both further research and development are required to bring this project to fruition.

We believe further refinement of the Capsicum primitives would be useful. Performance could be improved for sandbox creation, perhaps employing an Capsicum-centric version of the S-thread primitive proposed by Bittau. Further, a "logical application" construct might improve termination properties.

Another area for research is in integrating user interfaces and OS security; Shapiro has proposed that capability-centered window systems are a natural extension to capability operating systems. Improving the mapping of application security constructs into OS sandboxes would also significantly improve the security of Chromium, which currently does not consistently assign web security domains to sandboxes.

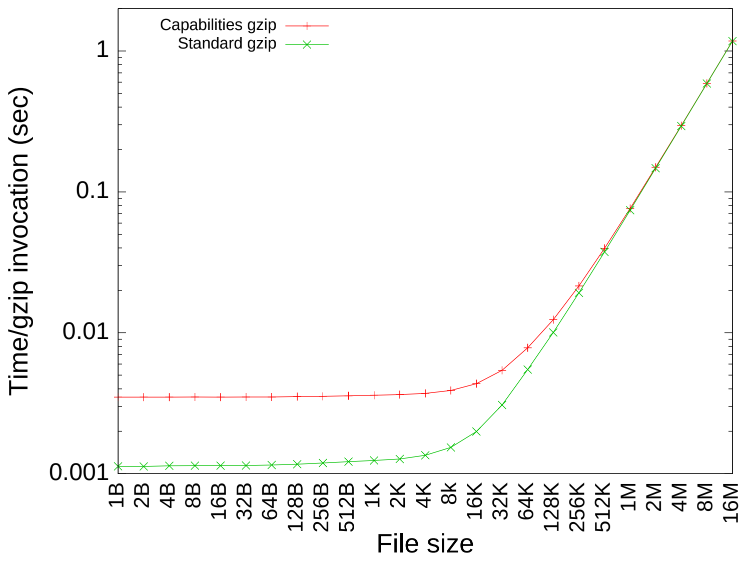

Figure 9: Run time per `gzip` invocation against random data, with varying file sizes; performance of the two versions of `gzip` come within 5% of one another at around a 512K file size for random data.

8 Related work

In 1975, Saltzer and Schroeder documented a vocabulary for operating system security based on on-going work on MULTICS [19]. They described the concepts of capabilities and access control lists, and observed that in practice, systems combine the two approaches in order to offer a blend of control and performance. The last 35 years of research have explored these and other security concepts, but the themes remain topical.

8.1 Discretionary and Mandatory Access Control

The principle of discretionary access control (DAC) is that users control protections on objects they own. While DAC remain relevant in multi-user server environments, the advent of personal computers and mobile phones has revealed the weakness of DAC as a protection model: on a single-user computer, all the eggs are kept in one basket. Section 5.1 demonstrates the difficulty of using DAC for malicious code containment.

Mandatory access control allows systemic enforcement of policies representing the interests of system implementers and administrators, and fall into two general categories. Information flow policies rely on tagging of subject and objects in the system with confidentiality and integrity labels—a fixed set of rules control information leakage then controls whether reads and writes are permitted. Multi-Level Security (MLS), formalized as Bell-LaPadula (BLP), protects confidential information from those who are not authorized to see it [3]. MLS's logical dual, the Biba integrity policy, implements a similar scheme protecting the integrity of systems against corruption, and is frequently used to protect Trusted Computing Bases (TCBs) [4].

MAC policies are robust against the problem of *confused deputies*, authorised individuals or processes who can be tricked into revealing confidential information. In reality, however, these policies are highly inflexible, requiring administrative intervention to change, which precludes the possibility of browsers creating isolated and ephemeral sandboxes "on demand" for each web site that is visited.

Type Enforcement (TE) in LOCK [20] and, later, SELinux [12] and SEBSD [25], offers greater flexibility by allowing arbitrary labels to be assigned to subjects (domains) and objects (types), and a set of rules to control their interactions. As demonstrated in Section 5.4, however, the requirement for administrative intervention and lack of a facility for ephemeral sandboxes limits applicability for applications such as Chromium: policy, by design, cannot be modified by users or software authors. Extreme granularity of control is under-exploited, or perhaps even discourages, highly granular protection—for example, the Chromium SELinux policy conflates different sandboxes allowing undesirable interference.

8.2 Capability systems, micro-kernels, and compartmentalisation

The development of capability systems has been tied to mandatory access control since inception, as capabilities were considered the primitive of choice for mediation in trusted systems. Neumann et al's Provably Secure Operating System (PSOS) [16], and successor LOCK, propose a tight integration of the two models, with the later refinement that MAC allows revocation of capabilities in order to enforce the *-property [20].

Despite experimental hardware such as Wilkes' CAP computer [27], the eventual dominance of general-purpose virtual memory as the nearest approximation of hardware capabilities lead to exploration of object-capability systems and micro-kernel design. Systems such as Mach [2], and later L4 [11], epitomize this approach, exploring successively greater extraction of historic kernel components into separate tasks. Trusted operating system research built on the trend through projects blending mandatory access control with micro-kernels, such as Trusted Mach [6], DTMach [22] and FLASK [24]. Micro-kernels have, however, been largely rejected by commodity OS vendors in favour of higher-performance monolithic kernels.

Micro-kernel-centric MAC has spread, without the benefits of enforced reference monitor separation, to commodity UNIX systems in the form of SELinux [12]. Despite a lack of deployment of capabilities, the other key security element to micro-kernel systems, research has continued in the form of EROS [23] (now CapROS), inspired by KEYKOS [9]. These systems, however, have not seen wide deployment.

OpenSSH privilege separation [17] and Privman [10] rekindled interest in micro-kernel-like compartmentalization of UNIX software on monolithic kernels, including projects such as the Chromium web browser [18] and Capsicum's logical applications. In fact, large application suites compare formidably with the size and complexity of monolithic kernels: the FreeBSD kernel is composed of 3,869,000 lines of C, whereas Chromium and WebKit come to a total of 4,136,000 lines of C++. How best to decompose monolithic applications remains an open research question, but Bittau's Wedge offers a particularly promising avenue of research in automated identification of software boundaries through dynamic analysis [5].

Seaborn and Hand have attempted to bring stronger application compartmentalisation primitives to UNIX through capability-centric Plash [21], and Xen-centric [15] approaches, respectively. The former, however, is built on the same weak UNIX primitives analysed in Section 5, and the latter suffers from similar problems to `seccomp`, in that the run-time environment for sandboxes is functionality-poor. Garfinkel's Ostia [7] also considers a capability-centric approach, focusing on delegation, but remains preoccupied with providing sandboxing as an extension, rather than a core OS facility.

A final branch of capability-centric research is capability programming languages. Java and the JVM have offered a vision of capability-oriented programming: a language run-time in which references and byte code verification don't just provide implementation hiding, but also allow application structure to be mapped directly to protection policies [8]. More specific capability-oriented efforts are E [13], the foundation for Capdesk and the DARPA Browser [26], and Caja, a capability subset of the JavaScript language [14].

9 Conclusion

We have described Capsicum, a practical capabilities extension to the POSIX API, and a prototype based on FreeBSD, which is planned for inclusion in FreeBSD 9.0. Our goal has been to address the needs of application authors who are already experimenting with sandboxing, but find themselves building on sand when it comes to effective containment techniques. We have discussed our design choices, contrasting approaches from research capability systems, as well as commodity access control and sandboxing technologies, but ultimately leading to a new approach. Capsicum lends itself to adoption by blending immediate security improvements to current applications with the long-term prospects of a more capability-oriented future. We illustrate this through adaptations of widely-used applications, from the simple `gzip` to Google's highly-complex Chromium web browser, showing how firm OS foundations make the job of application writers easier. Finally, security and performance analyses show that improved security is not without cost, but that the point we have selected on a spectrum of possible designs improves on the state of the art.

Acknowledgments We gratefully acknowledge sponsorship of this project by Google, Inc. We would also like to thank Mark Seaborn, Andrew Moore, Joseph Bonneau, Saar Drimer, Bjoern Zeeb, and Andrew Lewis for their helpful feedback on our APIs, prototype, and paper, and Sprewell for his work on porting Chromium to FreeBSD.

References

[1] The Chromium Project: Design Documents: OS X Sandboxing Design. `http://dev.chromium.org/developers/design-documents/sandbox/osx-sandboxing-design`.

[2] M. J. Acetta, R. Baron, W. Bolowsky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: a new kernel foundation for unix development. In *Proceedings of the USENIX 1986 Summer Conference*, pages 93–112, July 1986.

[3] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report 2547, MITRE Corp., March 1973.

[4] K. J. Biba. Integrity considerations for secure computer systems. Technical report, MITRE Corp., April 1977.

[5] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 309–322, 2008.

[6] M. Branstad and J. Landauer. Assurance for the Trusted Mach operating system. *Computer Assurance, 1989. COMPASS '89, 'Systems Integrity, Software Safety and Process Security', Proceedings of the Fourth Annual Conference on*, pages 103–108, 1989.

[7] T. Garfinkel, B. Pfa, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *Proc. Internet Society 2003*, 2003.

[8] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*.

[9] Norman Hardy. KeyKOS architecture. *SIGOPS Operating Systems Review*, 19(4), Oct 1985.

[10] Douglas Kilpatrick. Privman: A Library for Partitioning Applications. In *Proceedings of USENIX Annual Technical Conference*, pages 273–284, 2003.

[11] Jochen Liedtke. On microkernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, Copper Mountain Resort, CO, December 1995.

[12] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference table of contents*, pages 29–42, 2001.

[13] Mark S. Miller. The e language. `http://www.erights.org/`.

[14] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja: Safe active content in sanitized javascript, May 2008. `http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf`.

[15] Derek G. Murray and Steven Hand. Privilege Separation Made Easy. In *Proceedings of the ACM SIGOPS European Workshop on System Security (EUROSEC)*, pages 40–46, 2008.

[16] P. G. Neumann, R. S. Boyer, R. J. Geiertag, K. N Levitt, and L. Robinson. A provably secure operating system: The system, its applications, and proofs, second edition. Technical Report Report CSL-116, Copmuter Science Laboratory, SRI International, May 1980.

[17] Neils Provos, Markus Friedl, and Peter Honeyman. Preventing Privilege Escalation. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[18] Charles Reis and Steven D. Gribble. Isolating web programs in modern browser architectures. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 219–232, New York, NY, USA, 2009. ACM.

[19] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. In *Communications of the ACM*, volume 17, July 1974.

[20] O. Sami Saydjari. Lock: an historical perspective. In *Proceeedings of the 18th Annual Computer Security Applications Conference*. IEEE Copmuter Society, 2002.

[21] Mark Seaborn. Plash: tools for practical least privilege, 2010. `http://plash.beasts.org/`.

[22] E. J. Sebes. Overview of the architecture of Distributed Trusted Mach. *Proceedings of the USENIX Mach Symposium: November*, pages 20–22, 1991.

[23] Jonathan Shapiro, Jonathan Smith, and David Farber. EROS: a fast capability system. *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, Dec 1999.

[24] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Anderson, and Jay Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proc. 8th USENIX Security Symposium*, August 1999.

[25] C. Vance and R. Watson. Security Enhanced BSD. *Network Associates Laboratories*, 2003.

[26] David Wagner and Dean Tribble. A security analysis of the combex darpabrowser architecture, March 2002. `http://www.combex.com/papers/darpa-review/security-review.pdf`.

[27] M. V. Wilkes and R. M. Needham. *The Cambridge CAP computer and its operating system (Operating and programming systems series)*. Elsevier North-Holland, Inc., Amsterdam, The Netherlands, 1979.