ROBERT N.M. WATSON,
JONATHAN ANDERSON, BEN LAURIE,
AND KRIS KENNAWAY

introducing Capsicum: practical capabilities for UNIX

APPLICATIONS ARE INCREASINGLY turning to *privilege separation*, or *sandboxing*, to protect themselves from malicious data, but these protections are built on the weak foundation of primitives such as chroot and setuid. Capsicum is a scheme that augments the UNIX security model with fine-grained *capabilities* and a sandboxed *capability mode*, allowing applications to dynamically impose capability discipline on themselves. This approach lets application authors express security policies in code, ensuring that application-level concerns such as Web domains map well onto robust OS primitives. In this article we explain how Capsicum functions, compare it to other current sandboxing technologies in Linux, Mac OS, and Windows, and provide examples of integrating Capsicum into existing applications, from tcpdump and gzip to the Chromium Web browser.

Robert N.M. Watson is a PhD candidate at the University of Cambridge Computer Laboratory. His PhD research is in operating system security extensibility. Prior to joining the Computer Laboratory, he was a Senior Principal Scientist at McAfee Research, now SPARTA ISSO, where he directed commercial and government research and development projects in computer security, including the TrustedBSD MAC Framework now used for access control in FreeBSD, Juniper Junos, Mac OS X, and Apple iOS. His research interests include operating system security, network stack performance, and the evolving software-hardware interface. Mr. Watson is also a member of the board of directors for the FreeBSD Foundation, a 501(c)(3) non-profit supporting development of FreeBSD, a widely used open source operating system.

robert.watson@cl.cam.ac.uk

Jonathan Anderson is a PhD student in the University of Cambridge Computer Laboratory. His research interests include operating system security and privacy in distributed social networks.

jonathan.anderson@cl.cam.ac.uk

Ben Laurie works on security, anonymity, privacy, and cryptography and thinks object capabilities are the best hope we've got. He currently splits his time between the Applied Security group at Google and working on the Belay project at Google Research.

benl@google.com

Kris Kennaway is a Senior Site Reliability Engineer at Google. He is a former FreeBSD Security Officer and a former member of the FreeBSD Core Team. His interests include computer security, operating system scalability on multi-core hardware, and the design, care, and feeding of large-scale distributed systems. Kris received a PhD in theoretical physics from the University of Southern California in 2004.

kennaway@google.com

Compartmentalization

Today's security-aware applications are increasingly written as compartmentalized applications, a collection of cooperating OS processes with different authorities. This structure, which we term a "logical application" and illustrate in Figure 1, is employed to mitigate the harm that can be done if inevitable vulnerabilities in application code are exploited.

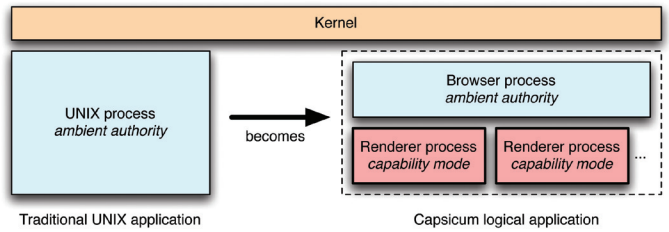

FIGURE 1: CAPSICUM HELPS APPLICATIONS SELF-COMPARTMENTALIZE.

For instance, in the Chromium Web browser, a malicious image that exploits a libpng vulnerability can be confined to a *renderer process* responsible for converting Web content such as HTML and compressed images into pixels. Such a process has less access to OS services such as the file system and network stack than the main *browser process*,

so the damage that can be done by malicious content is limited. Other widespread examples of software using this technique include PackageKit, Apple's Security Server, and OpenSSH's sshd.

Unfortunately, self-compartmentalizing code is very difficult to write, as contemporary commodity operating systems are firmly engrained with the notion of *ambient authority:* applications running with the full authority of the user who launched them. Creating a sandbox thus involves restricting existing access to user- or system-level rights, a process which frequently itself requires system privilege.

Capabilities

At the other end of the authority spectrum are capability systems, such as CMU's Hydra operating system [1], that support true least-privilege discipline in their applications. In such a system, application code can only exercise authority (e.g., access user files) through fine-grained *capabilities*, unforgeable tokens of authority, which have been delegated to it.

Capability systems are designed around delegation, since they allow tasks to selectively share fine-grained rights with other tasks through inheritance and explicit assignment. In this model, the operating system enforces the isolation of tasks and the restriction-associated capabilities, but semantically rich policy—what the capability *means* and who should have access to it—is defined by applications. This separation of mechanism and policy is very useful, and it is one which we sought to enhance on the UNIX platform by the addition of capability features.

Capsicum

Capsicum is a new approach to application compartmentalization. It is a blend of capability and UNIX semantics which, we believe, has some of the best characteristics of both. It allows applications to share fine-grained rights among several rigorously sandboxed processes, but preserves existing UNIX APIs and performance. Capsicum also provides application writers with a gradual adoption path for capability-oriented software design.

DESIGN

Capsicum extends, rather than replaces, standard UNIX APIs by adding new kernel primitives and userspace support code to help applications self-compartmentalize.

The most important new kernel primitives include a sandboxed *capability mode*, which limits process access to all global OS namespaces, and *capabilities*, which are UNIX file descriptors with some extra constraints. The userspace additions include *libcapsicum*, a library which wraps the low-level kernel features and a *capability-aware run-time linker.*

CAPABILITY MODE

Capability mode is a process credential flag set by a new system call, cap_enter, available to all users. 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 file system, PIDs and SystemV IPC namespaces.

Access to system calls in capability mode is also restricted: some system calls requiring global namespace access are unavailable, while others are constrained. For instance, sysctl can be used to 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 parameters as permitted in capability mode; all others are denied.

The system calls requiring constraints include sysctl, shm_open, which is permitted to create *anonymous memory objects*, but not named ones, and the openat family of system calls. The *at calls already accept a file descriptor argument as the directory relative to which to perform the open, rename, etc.; in capability mode, they are constrained so that they can only operate on objects "under" this descriptor. 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. This allows partial namespace delegation, as shown in Figure 2.

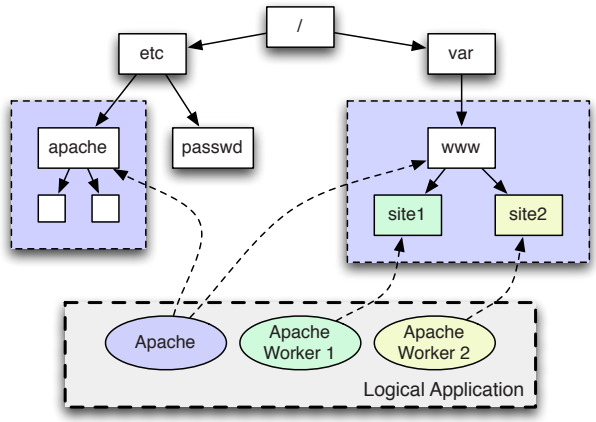

FIGURE 2: AUTHORITY OVER PORTIONS OF THE FILE SYSTEM CAN BE DELEGATED.

CAPABILITIES

In Capsicum, a capability is a type of file descriptor that wraps another file descriptor and constrains the methods that can be performed on it, as shown in Figure 3.

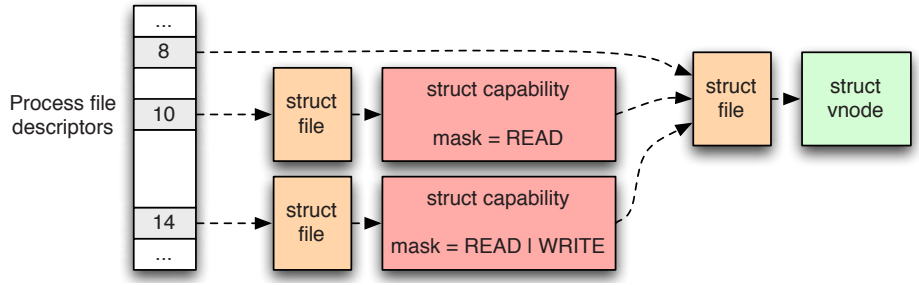

FIGURE 3: CAPABILITIES "WRAP" NORMAL FILE DESCRIPTORS, MASKING THE SET OF PERMITTED METHODS.

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 true object capabilities, however, they confer very broad rights as a side effect: even if a file descriptor is read-only, operations on metadata such as fchmod are

permitted. Capsicum restricts these operations by wrapping the descriptor in a capability descriptor, checking the mask of allowable operations whenever the file object is looked up. For instance, when the read system call is invoked with a capability, that capability can only be converted to a file object if its mask includes CAP_READ.

Capabilities are created via the cap_new system call, which accepts an existing file descriptor and a mask of rights as arguments. If the original descriptor is a capability, the result will be a new capability with a subset of the original's rights; applications may always reduce the privilege of a file descriptor, but they may never escalate it. Like other file descriptors, capabilities may be inherited across fork and exec, as well as passed via UNIX domain sockets.

There are approximately 60 rights which a capability can mask, striking a balance between pure message-passing (two rights: send and receive) and MAC systems (hundreds of access control checks). We have selected rights which align with logical methods on file descriptors; some system calls require multiple rights, and calls implementing semantically identical operations require the same rights. For example, pread (read to memory) and preadv (read to a memory vector) both require CAP_READ in a capability's rights mask, while read (read bytes using the file offset) requires CAP_READ|CAP_SEEK.

Capability rights are checked by fget, the in-kernel function for converting file descriptor numbers into in-kernel references. This strategy—implementing checks at a single point of service deep in the kernel, rather than in several system calls—is repeated throughout Capsicum, providing assurance that no alternate code paths exist which could be used to bypass checks.

Many past security extensions have composed poorly with UNIX security, leading to vulnerabilities. As a result, we disallow privilege elevation via fexecve using setuid and setgid binaries in capability mode. This restriction does not prevent setuid binaries from using sandboxes.

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. Processes, including libraries they use, may access resources with overly broad rights, or fail to relinquish access when it is no longer needed. Furthermore, introducing robust sandboxing forces fundamental changes to the UNIX run-time environment: even fork and exec rely on global namespaces—process IDs and the filesystem namespace.

libcapsicum therefore provides an API for starting sandboxed processes and ensuring that they only possess authority which has been explicitly delegated to them.

After creating a new process with the descriptor-oriented pdfork, libcapsicum cuts off the sandbox's access to global namespaces via cap_enter. In order to ensure that rights are not accidentally leaked from parent to child, it then closes all inherited file descriptors that have not been positively identified for delegation and flushes the address space via fexecve. Sandbox creation returns a UNIX domain socket that applications can use for inter-process communication (IPC) and for sharing additional rights between host and sandbox.

Starting a process inside a sandbox requires a Capsicum-aware run-time linker, which loads dynamic libraries from read-only directory descriptors rather than the global filesystem namespace. The main function of a program can call lcs_get to determine whether it is in a sandbox, retrieve sandbox state, query creation-time delegated capabilities, and retrieve an IPC handle so that it can process RPCs and receive runtime delegated capabilities. This allows a single binary to execute both inside and outside of a sandbox, diverging its behavior based on its execution environment.

APPLICATIONS

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. Capsicum does not solve this problem; what it does do is make it easy for an application writer, having decided where a security boundary should lie, to enforce it by creating a robust sandbox and sharing fine-grained, least-privileged rights with it.

We describe in this article two applications that we have modified to take advantage of Capsicum's features, one small and conceptually simple, tcpdump, and one large and complex, Chromium. For more case study details, please see our 2010 USENIX Security Symposium paper [2].

TCPDUMP

tcpdump provides an excellent example of Capsicum primitives offering immediate security benefits through straightforward changes. Historically, tcpdump has been a breeding ground for serious security vulnerabilities, as it has both root privilege and complex packet-parsing code. It is also a very simple program, however, which lends itself handily to sandboxing: resources are acquired early with ambient system privilege, after which packet processing depends only on open file descriptors.

True privilege dropping for tcpdump is accomplished with eight lines of code, shown in Figure 4. Verifying that unneeded privileges have been dropped can be done with the procstat tool; Figure 5 shows that the rights on STDIN have been appropriately constrained.

```
@@ -1197,6 +1199,14 @@
                        (void)fflush(stderr);
        }
 #endif /* WIN32 */
+       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 STDIN_FILENO");
+       if (lc_limitfd(STDERR_FILENO, CAP_FSTAT | CAP_SEEK | CAP_WRITE) < 0)
+               error("lc_limitfd: unable to limit STDERR_FILENO");
+       if (cap_enter() < 0)
+               error("cap_enter: %s", pcap_strerror(errno));
        status pcap_loop(pd, cnt, callback, pcap_userdata);
        if (WFileName =NULL) {
```

FIGURE 4: TCPDUMP DROPS ALL UNNEEDED PRIVILEGE WITH EIGHT LINES OF CODE.

```
PIDCOMM          FD  T  FLAGS      CAPABILITIES  PRO  NAME
1268 tcpdump     0   v  rw------c  fs            -    /dev/pts/0
1268 tcpdump     1   v  -w------c  wr,se,fs      -    /dev/null
1268 tcpdump     2   v  -w------c  wr,se,fs      -    /dev/null
1268 tcpdump     3   v  rw-------               - -  /dev/bpf
```

FIGURE 5: PROCSTAT -FC DISPLAYS CAPABILITIES HELD BY TCPDUMP. IN THE CASE OF STDIN, ONLY FSTAT (FS) IS PERMITTED.

CHROMIUM

Google's Chromium Web browser already uses a compartmentalized multi-process architecture similar to a Capsicum logical application on several operating systems [3], so it is an excellent platform for comparing Capsicum with other sandboxing techniques.

Once the FreeBSD port of Chromium was modified to use POSIX rather than System V shared memory (the former, from the Mac OS X port, is descriptor-oriented and thus permitted in Capsicum sandboxes), approximately 100 additional lines of code were required to limit access to file descriptors inherited by and passed to sandbox processes and to call cap_enter.

The result was a robust sandbox that, unlike porous approaches which require hundreds of lines of handcrafted, security-critical assembly code, could be completed in just two days.

Comparison

A plethora of existing security technologies have been used to construct sandboxes in security-aware applications such as Chromium. Each technology has its place—we do not claim that UNIX users and system integrity policies are obsolete—but each also has significant limitations when used for application sandboxing.

We compare Capsicum with five sandboxing mechanisms already employed by Chromium (see Table 1). Each mechanism is used to split the browser into a main browser process, which draws the browser's chrome and interacts with objects such as files, and several renderer processes, which execute untrusted code to uncompress images, interpret JavaScript, etc.

Operating system	*Model*	*Line count*	*Description*
Windows	ACLs	22,350	Windows ACLs and SIDs
Linux	chroot	605	SUID-root sandbox helper
Mac OS X	Seatbelt	560	Path-based MAC sandbox
Linux	SELinux	200	Type Enforcement sandbox domain
Linux	seccomp	11,301	seccomp and userspace syscall wrapper
FreeBSD	Capsicum	100	Capsicum sandboxing using cap_enter

TABLE 1: SANDBOXING MECHANISMS EMPLOYED BY CHROMIUM

Of the six mechanisms employed by Chromium, two are rooted in Discretionary Access Control (users and permissions), two in Mandatory Access Control (labels and system policies), and two in capabilities (unforgeable tokens of authority which are passed between or inherited by processes).

DISCRETIONARY ACCESS CONTROL

In Discretionary Access Control (DAC), the owners of objects specify what rights other users have on those objects; one common example of DAC is the UNIX permissions scheme. Such protections can be used to constrain application behavior if code runs with the authority of a user—such as "nobody" in traditional UNIX systems—with less privilege than the user running the application.

Chromium uses DAC to construct sandboxes on both Windows and Linux. In both cases, inter-user mechanisms fail to provide effective intra-user protections: the robustness of the sandbox is limited, because every user possesses some ambient authority.

Windows ACLs

On Windows, Chromium uses access control lists (ACLs) and security identifiers (SIDs) to effectively run renderer processes as an anonymous user who cannot access objects which belong to "real" users [3]. The unsuitability of the approach is demonstrated well; the model is both incomplete and unwieldy.

The approach is incomplete because objects which are not associated with any user do not receive the protections afforded to objects with ACLs. Some workarounds are possible—for instance, an alternate, invisible desktop is used to protect the user's GUI environment—but many objects remain completely unprotected, including FAT file systems on USB sticks and TCP/IP sockets. Thus, a "sandboxed" renderer process can communicate with any server on the Internet, or even the user's Intranet via a configured VPN!

The approach is also unwieldy in that many legitimate system calls by the sandbox are denied, and must be forwarded to a trusted process which services them on the sandbox's behalf. This forwarding, filtering, and servicing code comprises most of the 22,500 lines of code in the Windows sandbox module, and all of it is absolutely security-critical.

chroot

Chromium's suid sandbox on Linux also attempts to create a privilege-free sandbox using legacy DAC-based access control; the result is similarly porous, and it brings an additional requirement of system privilege.

In this model, access to the file system is limited to a virtual root directory via chroot, but access to other namespaces, including the network and System V shared memory (where the user's X window server can be contacted), is unconstrained.

This sandboxing mechanism also carries an additional requirement: system privilege is required to initiate chroot, so Chromium includes a SUID-root binary which is responsible for starting sandboxes. Thus, sandboxing can only be done with the permission of the system administrator, and any compromise of the setuid binary would have more disastrous consequences than the browser compromise it attempts to protect against.

MANDATORY ACCESS CONTROL

Mandatory Access Control (MAC) is used to enforce system policies such as "files labeled Top Secret shall only be read by users cleared to at least Top Secret," and "files labeled High Integrity shall only be modified by software labeled at least High Integrity."

MAC systems require policy to be described separately from application code. In the context of Multi-Level Secure systems and intelligence applications, this requirement allows rigorous and auditable control of information flow. In the context of sandboxing for consumer applications, however, it leads to the *dual-coding problem*: policy and code will get out of sync, especially if code is written by a vendor and policy by a distribution, so application writers must choose between false positives (legitimate actions are forbidden) and false negatives (illegitimate actions are permitted). In practice, very broad rights are often conferred to avoid blocking legitimate actions.

Furthermore, applying a MAC policy requires the involvement of the system administrator; in order to reduce application authority, system privilege is required. Users are, thus, only protected by MAC if the system administrator has already installed a policy for the software they run, and applications cannot dynamically reconfigure their sandboxes.

SELinux

Chromium supports MAC-based compartmentalization on Linux via an SELinux Type Enforcement policy [4]. We acquired such a policy, not from the Chromium repository, but from the Fedora project, a Linux distribution. Since code and policy come not just from different authors but from different organizations, the dual-coding problem may be expected to be severe.

Compounding the general dual-coding problem further, SELinux policies are so flexible and fine-grained that they are typically written using coarse-grained macros. As an example of one or both of these problems, the Fedora reference policy for Chromium assigns very broad rights, such as the ability to access the terminal device and read all files in /etc.

The requirement for system privilege in defining new policy and types means that Chromium cannot adapt its sandboxes to create new ephemeral security domains for each new website that is visited. For instance, Fedora's policy creates a single SELinux dynamic domain, `chrome_sandbox_t`, which is shared by all sandboxes. Thus, malicious code from evil.com is not prevented from interfering with the renderer process for bank.com.

Mac OS X Sandbox

Chromium also uses a MAC-based framework on Mac OS X to create sandboxes. The Mac OS X sandbox system allows processes to be constrained according to a Scheme-based policy language [5]. It uses the BSD MAC Framework [6] to check application activities against the compiled policy, which can express fine-grained constraints on the file system but, again, coarse all-or-nothing constraints on other namespaces, such as POSIX shared memory.

The Seatbelt-based sandbox model is less verbose than other approaches, but like all MAC systems, security policy must be expressed separately from code, which can lead to inconsistencies and vulnerabilities. Chromium's policy, while restricting access to the global filesystem namespace, allowed access to filesystem elements such as font directories.

CAPABILITIES

The third category of compartmentalization techniques contains *capability*-based approaches. As was mentioned above, capabilities are unforgeable tokens of authority which can be passed between processes, supporting a delegation-oriented security policy.

The UNIX file descriptor is an example of a capability-like object: an application cannot create one without the help of the OS kernel, and once created, it can be shared with other processes, which can then perform system calls such as read and write on it, even if those processes do not have permission to open the file for themselves. UNIX file descriptors are not well-formed capabilities, however. One serious problem with file descriptors is that they are very coarse: a descriptor may allow a process to fchmod the file it points to, even if it was opened with O_RDONLY. Thus, both of the following approaches further limit the authority that a file descriptor conveys and cut off ambient authority.

seccomp

One capability-oriented approach to sandboxing is Linux's seccomp. This is an optionally available mode which denies access to all system calls except read, write, and exit. Processes sandboxed in this way are quite rigorously confined, but only the very simplest applications can use the mode directly; in order to interact meaningfully with the user, network, file system, etc., significant scaffolding code is required to forward system calls, as in the case of the Windows sandbox. Like its Windows counterpart, the Chromium seccomp sandbox contains over a thousand lines of handcrafted, security-critical assembly code to set up sandboxing, implement system call forwarding, and craft a basic security policy (which, incidentally, is default-allow for all filesystem reads; a more complex policy would be even more unwieldy).

Capsicum

Capsicum brings capability concepts to UNIX, allowing sandboxes to be rigorously confined while still able to use capability-oriented UNIX APIs with full UNIX performance.

The modifications required to implement Chromium sandboxing on Capsicum are almost trivial—approximately 100 lines of code—yet they are more robust and flexible than other approaches which require hundreds or even tens of thousands of lines. Furthermore, in contrast to approaches that require system call interception and forwarding, sandboxed processes can operate on file descriptors, and the objects like shared memory which they refer to, with almost no performance degradation.

PERFORMANCE

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 goal is slightly different: application writers have already accepted significant overheads in order to adopt compartmentalization, so we seek to significantly improve security while keeping comparable performance.

Capsicum's capability mode and capabilities are designed to offer native UNIX performance for common operations, as frequently performed operations such as read and write are performed directly on capabilities. Likewise, directory descriptor delegation allows whole UNIX subtrees to be delegated to sandboxes, avoiding message passing on file open in many common cases. This approach is, fundamentally, a hybrid approach, combining elements of the UNIX OS model with a capability system: UNIX would offer unfettered access to the entire file system with privilege, and a capability system might rely on message-passing interposition to filter namespaces, imposing message-passing overhead on common operations.

Detailed performance results, as well as discussion of trade-offs between security and performance, can be found in our USENIX Security paper [2], but suffice it to say that Capsicum primitives are generally as fast as, and sometimes faster than, current UNIX primitives. Performance remains a critical area of research, however; while Capsicum may be cleaner and more efficient for existing privilege-separated applications, adapting further applications will perpetuate current security vs. performance trade-offs. Finding new approaches to improving security performance in the UNIX model is a key concern going forward.

Conclusion

Capsicum is a blending of capability-oriented security with UNIX APIs and performance. Capsicum provides OS foundations that applications can use to compartmentalize themselves with stronger confinement properties and, in some cases, better performance than existing sandboxing techniques. Capsicum is not a replacement for Discretionary or Mandatory Access Control, but we believe that it is superior to them as a platform for application self-compartmentalization.

Much still remains to be done—in some ways, Capsicum is just a platform for more interesting research in systems, programming, and UI security—but we believe that this is a very promising first step.

The Capsicum API and FreeBSD-based prototype are both available today under a BSD license, and more information can be found at http://www.cl.cam.ac.uk/research/security/capsicum/. Capsicum is intended for inclusion in mainline FreeBSD 9.

ACKNOWLEDGMENTS

The authors wish to gratefully acknowledge our sponsors, including Google, Inc., the Rothermere Foundation, and the Natural Sciences and Engineering Research Council of Canada. We would further like to thank Mark Seaborn, Andrew Moore, Joseph Bonneau, Saar Drimer, Bjoern Zeeb, Andrew Lewis, Heradon Douglas, Steve Bellovin, and our anonymous reviewers for helpful feedback on our APIs, prototype, and paper, and Sprewell for his contributions to the Chromium FreeBSD port.

REFERENCES

[1] E. Cohen and D. Jefferson, "Protection in the Hydra Operating System," *SOSP '75: Proceedings of the Fifth ACM Symposium on Operating Systems Principles* (ACM, 1975), pp. 141–60.

[2] R.M. Watson, J. Anderson, B. Laurie, and K. Kennaway, "Capsicum: Practical Capabilities for UNIX," in *Proceedings of the 19th USENIX Security Sympoisum* (USENIX, 2010), pp. 29–45.

[3] C. Reis and S.D. Gribble, "Isolating Web Programs in Modern Browser Architectures," *EuroSys '09: Proceedings of the 4th ACM European Conference on Computer Systems* (ACM, 2009), pp. 219–32.

[4] P. Loscocco and S. Smalley, "Integrating Flexible Support for Security Policies into the Linux Operating System," *Proceedings of the FREENIX Track: USENIX Technical Conference* (USENIX, 2001), pp. 29–42.

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

[6] R. Watson, B. Feldman, A. Migus, and C. Vance, "Design and Implementation of the TrustedBSD MAC Framework," in *Proceedings of the 3rd DARPA Information Survivability Conference and Exhibition (DISCEX)* (IEEE, April 2003).