A Comparison of Unix Sandboxing Techniques

Why sandboxing is different from historic approaches to Unix security, how we got where we are, and how Capsicum compares with Linux’s seccomp(2) and OpenBSD’s pledge(2).

Today’s users need more protection than traditional Unix systems have been able to deliver. The authors of operating systems have traditionally had a great deal of interest in systemic notions of privilege (e.g., the authority to inject code into the kernel via modules), but the users of computing systems often require finer-grained models of access control (e.g., the ability to share a single contact or delegate management of a single calendar). Rather than protecting multiple users from each other, the operating systems of today’s end-user devices must protect single users from their applications and those applications from each other. Historic Unix-derived systems have not made this task easy; in some cases, the protection users need has not even been possible.

Protection was a first-class objective of early, general-purpose operating systems and the hardware on which they ran [Lamp69, And72, SS72]. This early focus led naturally to the exploration and design of rigorous, general-purpose protection primitives such as capabilities [DV66] and virtual memory [BCD69, Lamp69]. In the transition from Multics to Unix dominance, this focus was lost. The result was a highly portable operating system that would go on to dominate contemporary thinking about operating systems, but with security features primarily organized around one threat model: users attacking other users (including accidental damage done by buggy software under development). This security model—Discretionary

By Jonathan Anderson

This work has been sponsored by the Research and Development Corporation of Newfoundland and Labrador (contract S404.1822.101) and the NSERC Discovery program (RGPIN-2015-06648), the Defense Advanced Research Projects Agency (DARPA), and the Air Force Research Laboratory (AFRL) (contract FA8650-15-C-7558).
Access Control (DAC)—can be implemented with Unix owner/group/other permissions or with Access Control Lists (ACLs), but it does not provide adequate support for application sandboxing. FreeBSD, Linux, and MacOS eventually acquired frameworks for enforcing systemic security policies such as multi-level security and integrity enforcement [WFMV03, WCM+02, Wat13], collectively known as Mandatory Access Control (MAC). Such policies represent the interests of system owners and administrators and provide an additional dimension along which enforcement can be specified, but they are better-suited to tasks such as protecting high-integrity files from low-integrity data than to supporting sandboxing in unprivileged applications.

The goal of sandboxing is to protect users from their own applications when those applications are exposed to untrusted content. Complex applications are regularly exposed to content from malicious sources, often embedded within difficult-to-parse protocols and file formats. This is especially true on the Internet, where even the most basic use cases involve ASN.1 parsing (for TLS) as well as parsing documents, Web pages, images, and videos as well as interpreting scripts or encoding/decoding cookies. Even the humble `file(1)` command was patched in 2014 for vulnerabilities in its parsing code [SA14:16]. Once a process is compromised by malicious content, the goal of a sandboxing policy is to limit the potential for damage to a small set of known outputs.

For example, a compromised word processor may be able to corrupt its output files, but it should not be able to search through a user’s home directory for private keys or credit card details. Sandbox-specific features (or, as they are sometimes referred to, attack-surface–reduction features) such as FreeBSD’s Capsicum, OpenBSD’s `pledge(2)`, and Linux’s `seccomp(2)` have appeared comparatively recently; we compare their effectiveness below.

**Sandboxing with DAC/MAC**

Prior to the introduction of sandboxing features in commodity operating systems, valiant efforts were made to confine or sandbox applications with the tools that were available. These efforts met with varying degrees of success, depending on how well the designed security policy fit onto a discretionary or mandatory access control (DAC or MAC) model. The most successful applications of sandboxing required relatively small code changes to meet their security objectives, which tended to fit the coarse-grained security model of DAC or the system-security perspective of MAC. Less successful forays into sandboxing required thousands of lines of code, sometimes with large amounts of hand-crafted assembly, a requirement for system (superuser) privilege and—often—a failure to truly enforce the desired security policy.

An early—and relatively successful—implementation of sandboxing was Provos et al’s privilege separation of the OpenSSH server [PFH03]. This work used discretionary access control features to prevent a compromised SSH server process from exercising the privileges of the superuser. The SSH server requires superuser privilege in order to bind to TCP port 22, but it is desirable that a compromised SSH process not be able to access system resources such as the filesystem before a user has authenticated (i.e., the server should be put into a pre-auth sandbox); it is also desirable that the server post-authentication only be able to exercise the authority granted to the authenticated user (from within a post-auth sandbox). Provos et al split the SSH server process into a trusted monitor process that retained superuser privilege and untrusted child processes that would use the superuser privilege to drop privilege, changing their user and group IDs to those of unprivileged users. In the pre-auth sandbox, an SSH server process could run as the `nobody` user and have its root directory changed to an empty directory using the `chroot(2)` system call. In the post-auth sandbox, a process would have its UID/GID changed to those of the authenticated user. This approach to sandboxing was successful for two reasons:

1. **User-oriented policy:** The goal of SSH privilege separation is to keep compromised processes from exercising the authority of the superuser. This policy goal aligns well with the Unix DAC model: it can be expressed entirely in terms of UIDs, GIDs, and filesystem directories with Unix permissions. The policy does not protect a user from misbehavior post-authentication: it protects the system and other users.

2. **Extant privilege:** Operations such as changing a process’s UID or root directory require superuser privilege, which the SSH...
server undergoing privilege separation already possessed. The `sshd` process was already a security-critical piece of software run as `root`: privilege separation was a monotonic decrease in authority. This is not the case for the more general case of sandboxing, however: it is undesirable to require unprivileged software to run as `root` in order to drop privilege.

At the other end of the spectrum, we have previously compared several DAC- and MAC-based approaches to sandboxing render processes in the Chromium web browser [WALK10]. In that work, we found that DAC and MAC mechanisms were a poor fit for the application compartmentalization use case. DAC is designed to protect users from each other, but in the case of a Web browser—or any other sophisticated, multiprocess user application—the security goal is to limit the damage that can be done by a rogue process after it is compromised by untrusted content. DAC alone cannot control access to unlabeled objects such as System V shared memory (in Linux) or FAT filesystems (in Windows). As with OpenSSH, `chroot(2)` can be used to put a process into an environment of limited filesystem access, but unlike OpenSSH, the superuser privilege required to use `chroot(2)` is not naturally found in Web browsers (or office suites, music players, other desktop applications, etc.). Thus, in order to avail of the DAC-based protection that did exist, portions of the application had to be shipped with the `setuid` bit set on a `root`-owned binary!1

Mandatory Access Control (MAC) is also a poor fit for application sandboxing. It requires a dual coding of policy: once in the code that describes what the application does and once in a separate policy that describes what the application is allowed to do. The SELinux policy from our original Capsicum comparison involved thousands of lines of policy, but even modern AppArmor profiles encoded in a domain-specific language can require hundreds of lines of subtle and complex policy, not including policy elements included from system policy libraries (e.g., `#include <abstractions/ubuntu-browsers.d/java>`). It can be very difficult to write and maintain complex MAC policies, with failures in functionality (e.g., a lack of access to `/proc/[0-9]*/oom_score_adj`) being more obvious to

---

1 Today’s Chrome no longer uses the DAC-based sandbox on Linux, but the above comments about `chroot(2)` and privilege still apply to the new sandboxing model.

---

ZFS experts make their servers

Now you can too. Get a copy of.....

Choose ebook, print, or combo. You’ll learn to:

• Use boot environment, make the riskiest sysadmin tasks boring.
• Delegate filesystem privileges to users.
• Containerize ZFS datasets with jails.
• Quickly and efficiently replicate data between machines.
• Split layers off of mirrors.
• Optimize ZFS block storage.
• Handle large storage arrays.
• Select caching strategies to improve performance.
• Manage next-generation storage hardware.
• Identify and remove bottlenecks.
• Build screaming fast database storage.
• Dive deep into pools, metaslabs, and more!

Link to: http://zfsbook.com

WHETHER YOU MANAGE A SINGLE SMALL SERVER OR INTERNATIONAL DATACENTERS, SIMPLIFY YOUR STORAGE WITH FreeBSD Mastery: Advanced ZFS. GET IT TODAY!
developers than lapses in protection (e.g., allowing access to /usr/bin/xdg-settings, which relies on the PATH environment variable not being hijacked). This complexity is a hint that we are attempting to fix the square peg of sandboxing into the round hole of MAC.

"Sandboxing" with System-call Interposition

Before comparing today’s approaches to sandboxing, it is essential to understand an intermediate approach that was attempted in the late 1990s and early 2000s. This approach was attractively simple, but ultimately failed to provide the security benefits it advertised. Those who fail to learn from this now-discredited approach are condemned to repeat its mistakes in their "new" approaches to application sandboxing.

A seminal attempt to generalize policy enforcement for arbitrary applications without system privilege was Fraser, Badger and Feldman’s Generic Software Wrappers [FBF00], which inspired systems such as Provos’s systrace [Prov03]. These system-call interposition systems used userspace wrappers or shallow modifications to the system-call layer of an OS kernel to intercept system calls. Once intercepted, these calls’ arguments could be inspected and a policy decision could be made as to whether or not the call should be allowed. For example, instead of using chroot(2) to limit a process’s access to the filesystem, every open(2) call could be inspected and the filename argument could be compared against a whitelist of paths the process was allowed to open. System call policies could be described in languages that, while requiring dual coding as in MAC, had the benefits of concision and comprehensibility, as shown in Figure 1. System call wrappers had the twin benefits of being relatively simple to implement and relatively simple to use. Unfortunately, their simplicity translated into a failure to engage with the complexities of concurrent accesses in operating systems, as demonstrated by Watson in 2007 [Wat07].

Objects named by Unix system calls are concurrent on multiple levels. At the shallowest level, all of a process’s threads are contained within the same virtual address space and can thus manipulate the same data—this includes strings being passed as arguments to system calls. When system call wrappers work in userspace, a malicious process can submit a system call for execution with a path that is known to be whitelisted and then, while the wrapper’s policy check is executing, modify the value of the memory containing the filename to a different path. Thus, the path that is checked against the policy can be different from the path that is eventually accessed. To use Watson’s language, this is a Time-of-check-to-time-of-use (TOCTTOU) vulnerability [Wat07].

TOCTTOU vulnerabilities are not merely found at this shallow layer of interception, however. If they were, interception would only need to be done via RPC to be secure. System call wrappers are vulnerable in a deeper, more fundamental way: even if the name used to reach an OS object such as a file remains constant, the meaning of that name can change. Path lookup is an incremental operation in Unix: looking up a file named /home/jon/foo.txt will involve interactions with at least four vnodes in a virtual filesystem (the root node, two directories and the file itself). While each individual lookup (e.g., retrieving the jon directory entry) must be done with due care for

```bash
# GSWTK policy
#include ".../wr.include/platform.ch"

wrapper bsd_noadmin {  
  bsd:op[mount ll unmount ll ...] pre {  
    return WR_DENY | WR_BADPERM;
  }
}

# systrace policy
Policy: /usr/sbin/named, Emulation: native
  native__sysctl: permit
  native-accept: permit
  native-bind: sockaddr match "inet-:*:53" then permit
  native-break: permit
  native-chdir: filename eq "/" then permit
  native-chdir: filename eq "/namedb" then permit
  native-chroots: filename eq "/var/named" then permit
  native-close: permit
  native-connect: sockaddr eq "/dev/log" then permit
...
```

Fig. 1
Policies governing system call wrapper behavior for the Generic Software Wrapper Toolkit and systrace (reproduced from [FBF00] and [Prov03], respectively).
concurrency, e.g., while holding a lock, the overarching path lookup is not an atomic operation. A path is a list of instructions, not a name. While one process is walking a directory hierarchy, another can be changing the filesystem, moving files, moving directories, even changing symbolic links. System-call interposition, even if performed via RPC with no possibility of in-memory path substitution, cannot guarantee that the file named by a path at the time a policy decision was made is the same file that will be looked up by the system call doing the lookup.

Fundamentally, the weakness of system-call interposition is that its policy decisions (i.e., checks) are not made atomically with the effects of those decisions. This is not a vulnerability that requires a patch, but a fundamental limitation of the approach; it is why such methods are no longer used on contemporary operating systems (OpenBSD expunged systrace in April of last year [Gros16]). However, even though system-call interposition systems have been deprecated, the underlying concept returns to haunt more modern sandboxing frameworks.

A Comparison of Sandboxing Frameworks

More recently, open-source Unix derivatives have implemented new frameworks to aid in application sandboxing. These frameworks include, most comparably, Linux’s seccomp(2), OpenBSD’s pledge(2), and FreeBSD’s Capsicum (cap-

sicum(4)). Although they were all created with the goal of enabling simple sandboxing, they have achieved varying degrees of success.

Linux: seccomp(2)

Since 2005, Linux has included a feature called "secure computing mode," or seccomp(2) for short [Cor09]. The original version of seccomp(2) provided a strong, comprehensible security policy: processes in "secure computing mode" can use the read(2) and write(2) system calls to operate on files they have previously opened (or had delegated to them), sigreturn(2) to support signal delivery, and the exit(2) system call to terminate the process. It is simple for a process to enter seccomp mode, as shown (in abridged form\(^3\)) in Figure 2. This policy had the benefit of clarity and it did permit processes to operate as filters performing otherwise-pure computation, but very few applications are able to perform meaningful work within such a restrictive sandbox. For example, we previously found that Chrome’s use of the "pure" seccomp(2) mode required over a thousand lines of security-critical assembly-language code to forward system calls outside of the sandboxed process and into a trusted process that would perform the system calls on its behalf [WALK10].

To provide a richer environment for computing, modern seccomp(2) allows programs to specify their own security policy beyond the four system calls enumerated above. In this new version of seccomp(2), a process can specify a program to check each system call’s validity before executing it. This program is written in the BPF bytecode format. The BSD Packet Filter (BPF) [MV93], inspired by the CMU/Stanford Packet Filter (CSPF), itself inspired by earlier work on the Xerox Alto [MRA87], is a virtual machine that interprets bytecode. It was originally designed to facilitate high-performance networking by allowing userspace processes to describe a filter for the kernel to apply to network packets without giving up the safety of the kernel/user mode separation. When applied to seccomp(2), BPF provides a syntax for describing programs that check system calls within the Linux system call handler.

An example of a simple system-call whitelisting filter is shown in Figure 3. This illustrates the extreme flexibility and programmability of seccomp-bpf: almost any check that can be imagined on a system call’s arguments can be expressed in an assembly-like language like BPF. However, the corollary to this is that because anything can be checked by the programmer,

\[\text{if (prctl(PR_SET_SECCOMP, SECCOMP_MODE_RESTRICT) != 0)}\]
\[\{\]
\[\text{err(-1, "error entering secure computing mode");}\]
\[\}\]

Fig. 2 Entering the original, "pure" version of Linux’s secure computing mode is trivial. Once a process is in seccomp mode it can never leave.

2 Discussion of Apple’s Sandbox framework and its MAC Framework underpinnings is left to other sources [Wat13].
3 Full source code for the examples in this section can be found at https://github.com/trombonehero/sandbox-examples.
everything must be checked by the programmer. In order to build a meaningful whitelist of system calls, not only must the offset of the syscall number within a larger structure be exposed to user-mode programs, the provided filter must also inspect the current architecture in order to interpret the syscall number (Linux uses different system call numbers on different architectures). Furthermore, as semantics are left to the programmer, it is possible—indeed, all too easy—to construct inconsistent system call policies that deny some operations while allowing equivalent operations to be performed. For example, the policy in Figure 3 does not allow unrestricted open(2) calls, but it permits openat(2), which can be made to behave equivalently to open(2). A seccomp-bpf filter is intimately tied to the details of the program whose behavior it filters, making it the responsibility of the application authors, but constructing a seccomp-bpf system call filter requires meticulous attention to the sorts of details (assembly programming in BPF opcodes, layouts and semantics of Linux kernel syscall handling structures) that are entirely outside of most application authors’ experience and working knowledge.

Beyond simple syscall whitelists, seccomp-bpf is both more complex and more problematic. It is possible to construct seccomp-bpf filters on system call arguments such as filenames, but as with GSWTK and systrace, it is impossible to check paths meaningfully at the system-call handling layer. A program may be permitted to access /var/tmp/*, but if /var/tmp/foo is a symbolic link that can be updated in a race with the BPF filter, what policy has truly been enforced? The openat example at https://github.com/trombonehero/sandbox-examples demonstrates how a process restricted using seccomp-bpf can escape from its intended bounds, in this case creating files outside of an application’s intended working directory.

For all of these reasons, seccomp-bpf alone is insufficient to truly sandbox arbitrary application code. A complete application sandbox must also use the Linux clone(2) system call to sequester a process within a new IPC namespace (to cut off access to the host’s global System V IPC namespace), network namespace (interfaces, routing, firewall, /proc and /sys/class/net, etc.), mount namespace (similar to chroot(2)), and PID namespace (to cut off inappropriate uses of kill(2)). Creating such namespaces requires the CAP_SYS_ADMIN privilege, which is effectively equivalent to superuser privilege on Linux. Thus, creating an effective application sandbox on Linux requires running programs as root or creating setuid binaries.

OpenBSD: pledge(2)
Since v5.9 was released in 2016, OpenBSD has shipped with pledge(2), a mechanism for putting a process into a "restricted-service operating mode". The manual page for pledge(2) does not describe it as a security mechanism, but other communications by its developers do. The essence of pledge(2) is a simpler, more flexible alternative to seccomp-bpf.
more easily used take on the seccomp(2) concept. Instead of defining a BPF program to filter out system calls, pledge(2) groups system calls into categories such as stdio (which includes read(2), write(2), dup(2), and clock_getres(2)) and rpath (which allows read-only filesystem effects from chdir(2), openat(2), etc.). It is possible to make a pledge with the empty string, in which case no further system calls but _exit(2) are permitted, but this can result in processes aborting when atexit(3) code triggered by C startup routines in _start() call mprotect(2) on libc.

pledge(2) is considerably simpler to use than equivalent seccomp-bpf functionality. Figure 4 shows an example of pledge(2) use that applies a system call filter to the current process using

```c
if (pledge("stdio rpath cpath flock", NULL) < 0) {
    err(-1, "error in pledge()");
}
```

Fig. 4 A system call filtering policy is considerably simpler to install with pledge(2) than with seccomp-bpf.

more system calls than that of Figure 3. However, as with seccomp-bpf, this simple, superficial filtering of system calls provides illusive security guarantees. The provided system call categories may usefully describe the requirements of trivial OpenBSD base system applications, but for complex applications, categories such as wpath are effectively meaningless. If an application needs to open private files for writing, then wpath must be "pledged," but wpath also authorizes opening any file on the filesystem with the correct DAC mode for writing. Unlike seccomp-bpf, pledge(2) makes policy construction simple, but like its Linux analog, it makes the construction of inconsistent or meaningless policies easy to do by default.

The pledge(2) system call also takes a paths argument containing a whitelist of allowable paths, but that functionality has been marked as "unavailable" in the pledge(2) manual page since early 2016 [Pled17]. Were it available, the shallow whitelisting functionality would suffer from the same TOCTTOU vulnerabilities as systrace and seccomp-bpf. However, the greatest weakness of pledge(2) is that a compromised process can disable the security mechanism if the original pledge(2) call included the exec system-call category. Despite claims that "abilities can never be regained" [Pled17] and "in OpenBSD, once a mitigation is working well, it cannot be disabled" [deRa15], pledge(2) does not have the one-way property of seccomp(2) or capsicum(4). In those systems, a process that enters a restricted state remains there together with all of its subsequently-created children, but an OpenBSD process’s pledge(2)-restricted state is cleared on exec(2).

As with seccomp-bpf (and GSWTK/systrace before that), system call filtering with pledge(2) is insufficient to apply a meaningful security policy to applications more complex than read–compute–write filters. The difference is that, although it is the simpler framework to use, pledge(2) is not backed by clone(2)-based mechanisms for implementing more rigorous security policies. Thus, as with the now-discontinued-by-OpenBSD systrace, pledge(2) should be seen as a debugging and mitigation feature to catch unskilled adversaries rather than a rigorous mechanism on which to build security policies.

FreeBSD: capsicum(4)

The Capsicum compartmentalization framework is different from seccomp-bpf and pledge(2) in two key ways. First, Capsicum employs a principled, coherent model for restrictions on processes when applications are compartmentalized. This is implemented by Capsicum’s capability mode. Second, Capsicum employs fine-grained, monotonic reduction of authority on specific OS objects accessed via attenuated file descriptors, called capabilities.

Capability Mode

Like seccomp-bpf and pledge(2), capsicum(4) supports putting processes into a restricted mode in which system calls behave differently from "normal" processes. The key distinction is how the restrictions are chosen. Rather than a superficial focus on specific system calls, many of which have overlapping responsibilities and provide independent means of accomplishing the same objective, Capsicum focuses on a fundamental principle underlying them all: access to global namespaces.

In Capsicum, the cap_enter(2) system call causes a process to enter capability mode, in which all access to OS objects (files, sockets, processes, shared memory, etc.) must be done
through capabilities (described below) rather than using ambient authority. Ambient authority describes the normal authority of a process to act on behalf of its user, doing anything that the user is permitted to do by the Unix DAC model. This includes access to other processes via PID, files via path or NFS file handle, sockets via protocol addresses, shared memory via System V IPC name or POSIX shared memory path, etc. By contrast, a process in capability mode is not allowed to access any new resources via global namespaces (path, PID, protocol address, etc.). Resources represented by already-open file descriptors (or descriptors passed into a process via Unix message passing) are normally subject to restrictions described below (under "capabilities"). New file descriptors may also be derived from existing descriptors using system calls such as accept(2) or even openat(2), provided that only local names are used. In the case of openat(2), this requires that path search start relative to an already-open directory descriptor, not AT_FDCWD, and that path evaluation only traverse "down" inside a directory and not "up" via "..". This restriction on path lookup is enforced within FreeBSD's namei() function, deep within the kernel and atomic with the lookup being policed.

The policy enforced by Capsicum's capability mode is internally consistent, as it is based on a fundamental principle rather than shallow system call syntax. The policy enforced by Capsicum's capability mode is internally consistent, as it is based on a fundamental principle rather than shallow system call syntax. It can enforce the same restrictions as the limited, internally-consistent use cases of seccomp-bpf and pledge(2): if a process enters capability mode with no resources held but readable/writable file capabilities, no side effects can be caused on the system except those described by the descriptors. To enable more sophisticated behaviors, Capsicum provides capabilities to facilitate principled sharing of resources within a coherent security model.

Capabilities
The historic concept of a capability in computer science is that of an identifier for an object combined with operations that can be performed on it. This sense of the word was described by Dennis and Van Horn in the late 1960s [DV66], and its echoes can be heard in Unix today. In Dennis and Van Horn's conception, a capability was an index into a list of capabilities maintained by the supervisor on behalf of a process. This concept carried forward into Multics and then morphed into the file descriptor as we know it today in Unix [RT78]. Like capabilities, file descriptors are indices into a supervisor-maintained list of OS objects; they are also associated with operations that may be performed on them based on the flags they were opened with (e.g., O_RDONLY). Unlike capabilities, however, file descriptors carry unexpected, implicit authority with them that cannot be monotonically reduced. For example, an application cannot open(2) a descriptor with flag O_RDWR, dup(2) it, commute the new descriptor to a read-only descriptor, and share it with an untrusted worker process. Even when a file descriptor is opened read-only, the Unix DAC model will still permit system calls like fchmod(2) to—perhaps unexpectedly—manipulate file metadata.

Capsicum's implementation of capabilities provides for the monotonic reduction of fine-grained rights ("authorities") on specific objects. It does this by attaching rights such as CAP_READ, CAP_FSTAT, CAP_MMAP, CAP_FCHMOD, etc., to descriptors. These classes of behaviors correspond to methods on kernel objects and are related to sets of system calls that require them. For example, to open a read-write file relative to a directory with openat(2), that directory descriptor must have at least CAP_READ, CAP_WRITE, and CAP_LOOKUP enabled for it. Outside of capability mode, unsandboxed processes using ambient authority with system calls such as open(2) are returned file descriptors with all rights implicitly granted. This preserves compatibility with traditional Unix semantics while allowing for uniform enforcement of capability rights both inside and outside capability mode. Rights on descriptors can be attenuated using cap_rights_limit(2), descriptors can be inherited by or passed to sandboxed processes, and new descriptors derived
from existing ones (e.g., via accept(2) or openat(2)) derive their delegation with confidence.

Sandboxing with Capsicum

Capsicum allows application authors to apply rigorous security policy to their applications with—in some cases—a minimum of effort. Today, even moderately complex applications such as hypervisors and Web browsers can support rich use cases by opening resources (including resource-bearing resources such as directories and server sockets), limiting the rights associated with those resources and then entering capability mode. The work required to sandbox the bhyve hypervisor in this way is shown in Figure 5. Efforts are ongoing to make the Capsicum model applicable to broader classes of applications, including applications that require access to external resources such as powerboxes [Yee04], even when they are oblivious to sandboxing features [AGW17].

Starting from a rigorous foundation, Capsicum is a platform that can support complex behaviors. Since its security policies are both simple and coherent, application authors can build supporting services on this foundation without requiring expertise in kernel internals or the fear of constructing an incoherent security policy. We therefore see Capsicum as a generative platform that enables application authors to focus on what they do best, using rigorous security-enabling tools without requiring extreme security expertise. It is our hope that providing authors with tools for safe software construction will enable future applications to better protect users, not just from each other, but from their own applications.

References


