hash=537f9fdb0b0514ca3165e9b622c979c9family=Daley, familyi=D., given=Robert C., giveni=R. C.

Towards Adaptive Process Confinement Mechanisms

COMP5900I Literature Review

William Findlay

December 13, 2020

Abstract

The *process confinement problem* or *sandboxing problem* has been studied for nearly five decades. The essence of the problem is confining a running process, restricting its behaviour to its intended functionality; if compromised, such a process would not be able to inflict meaningful damage on the rest of the running system. Despite a rich history of proposed solutions, this problem remains essentially unsolved to this day. I believe that the key to solving the process confinement problem is to move away from conventional approaches and instead approach the problem from an *adaptive security* perspective. At a high level, adaptive security involves creating security solutions that are widely deployable, highly transparent, and robust in the presence of attacker innovation.

In this literature review, I define precisely what an *adaptive process confinement mechanism* is and examine and critique the current process confinement literature using this adaptive definition. In doing so, I find that the vast majority of existing process confinement mechanisms are incompatible with this adaptive definition, although some are certainly closer than others. I conclude by examining what the move towards a truly adaptive process confinement mechanism might look like.

1 Introduction

Restricting unprivileged access to system resources has been a key focus of operating system security research since the inception of the earliest timesharing computers in the late 1960s and early 1970s [13, 29, 43]. In its earliest and simplest form, access control in operating systems meant preventing one user from interfering with or reading the data of another user. The natural choice for many of these early multi-user systems, such as Unix [43], was to build access control solutions centred around the user model—a design choice which has persisted in modern Unix-like operating systems such as Linux, OpenBSD, FreeBSD, and macOS. Unfortunately, while user-centric permissions offer at least some protection from other users, they fail to protect users from *themselves* or from their own *processes*. Researchers long ago recognized that finer granularity of protection is required to restrict a process to its desired functionality [34]. This is often referred to as *the process confinement problem* or *the sandboxing problem*.

Despite decades of work since Lampson's first proposal of the process confinement problem in 1973 [34], it remains largely unsolved to date [17]. This begs the question as to whether our current techniques for process confinement are simply inadequate for dealing with an evolving technical and adversarial landscape. Here, I argue that, to solve the process confinement problem, we need to rethink the status quo in process confinement and instead move towards *adaptive process confinement mechanisms*.

1.1 Defining Adaptive Process Confinement

Here, I define adaptive process confinement mechanisms as those which greatly help defenders confine their processes, are easily adoptable across a variety of system configurations, and are robust in the presence of attacker innovation. Roughly, this definition can be broken down into the following properties:

P1. Robustness to attacker innovation. An adaptive process confinement mechanism should continue to protect the host system, even in the presence of attacker innovation. That is, it should be resistant to an adaptive adversary.

P2. Adoptability. An adaptive process confinement mechanism should require minimal effort to adopt on a variety of system configurations. It should work out of the box on the majority of target systems and should be deployable in a production environment without major security or stability concerns.

P3. Reconfigurability. An adaptive process confinement mechanism should be highly reconfigurable based on the needs of the end-user and the environment in which it is running. This reconfiguration could either be automated, semi-automated, or manual, but should not impose significant adoptability barriers (c.f. P2).

P4. Transparency. An adaptive process confinement mechanism should be as transparent to the end-user as possible. It should not get in the way of ordinary system functionality, and should not require the modification of application source code to function. Further, its base functionality should not require significant user intervention, if at all.

P5. Usability (by non-experts). An adaptive process confinement mechanism should maximize its usability such that it is usable by the largest and most diverse set of defenders possible. In particular, it should not require significant computer security expertise from its users.

1.2 Outline of the Literature Review

To argue the case for adaptive process confinement, we need to understand the existing process confinement literature from an adaptive perspective. In this literature review, I examine the process confinement literature and critique its adaptiveness according to the definition outlined above (P1–P5). I then argue the case for moving towards truly adaptive process confinement mechanisms and explore what such a solution might look like.

The rest of this paper proceeds as follows. Section 2 presents the process confinement threat model. Section 3 examines the status quo in process confinement. Section 4 presents an adaptive analysis of the systems discussed in Section 3 and discusses the move toward truly adaptive process confinement. Section 5 concludes.

2 The Process Confinement Threat Model

To understand why process confinement is a desirable goal in operating system security, we must first identify the credible threats that process confinement addresses. To that end, I first describe three attack vectors (items A1 to A3), followed by three attack goals (items G1 to G3) which highlight just a few of the threats posed by unconfined processes to system security, stability, and user privacy.

A1. Compromised processes. Unconfined running processes have classically presented a valuable target for attacker exploitation. With the advent of the Internet, web-facing processes that handle untrusted user input are especially vulnerable, particularly as they often run with heightened privileges [11]. An attacker may send specially crafted input to the target application, hoping to subvert its control flow integrity via a classic buffer overflow, return-oriented programming [46], or some other means. The venerable Morris Worm, regarded as the first computer worm on the Internet, exploited a classic buffer overflow vulnerability in the `fingerd` service for Unix, as well as a development backdoor left in the `sendmail` daemon [51]. In both cases, proper process confinement would have

eliminated the threat by preventing the compromised programs from impacting the rest of the system.

A2. Semi-honest software. Here, I define semi-honest software as that which appears to perform its desired functionality, but which additionally may perform some set of unwanted actions without the user's knowledge. Without putting a proper external confinement mechanism in place to restrict the behaviour of such an application, it may continue to perform the undesired actions *ad infinitum*, so long as it remains installed on the host. As a topical example, an `strace` of the popular Discord [19] voice communication client on Linux reveals that it repeatedly scans the process tree and reports a list of *all applications* running on the system, even when the user has turned off the "display active game" feature[1]. This scanning behaviour represents a clear violation of the user's privacy expectations.

A3. Malicious software. In contrast to semi-honest software, malicious software is that which is expressly designed and distributed with malicious intent. Typically, this software would be downloaded by an unsuspecting user either through social engineering (e.g. fake antivirus scams) or without the user's knowledge (e.g. a drive-by download attack). In the case of a computer virus, malicious software may replicate itself on the host by infecting other (originally benign) binaries. It would be useful to provide the user with a means of running such potentially untrustworthy applications in a sandbox so that they cannot damage the rest of the system.

G1. Installation of backdoors/rootkits. Potentially, the most dangerous attack goal in the exploitation of unconfined processes is the establishment of a backdoor on the target system. A backdoor needn't be sophisticated—for example, installing the attacker's RSA public key in `ssh`'s list of authorized keys would be sufficient—however, the most sophisticated backdoors may result in permanent and virtually undetectable escalation of privilege. For instance, a sophisticated attacker with sufficient privileges may load a *rootkit* [6] into the operating system kernel, at which point she has free reign over the system in perpetuity (unless the user manages to somehow remove the rootkit or reinstalls the infected operating system).

G2. Information leakage. An obvious goal for attacks on unconfined processes (and indeed the focus of the earliest literature on process confinement [34]) is information leakage. An adversary may attempt to gain access to personal information or other sensitive data such as private keys, password hashes, or bank credentials. Depending on the type of information, an unauthorized party may not even necessarily require elevated privileges

[1] This feature allows Discord to report, in the user's status message, what game the user is currently playing. The "display active game" feature appears to be the original motivation behind scanning the process tree.

to access it—for instance, no special privileges are required to leak the list of processes running on a Linux system (as in the case of Discord [19] highlighted above).

G3. Denial of service. A compromised process could be used to mount a denial of service attack against the host system. For example, an attacker could take down network interfaces, consume system resources, kill important processes, or cause the system to shut down or reboot at an inopportune moment.

As shown in the examples above, unconfined processes can pose significant threats to system security and stability as well as user privacy. The advent of the Internet has exacerbated many of these threats. Unconfined network-facing daemons continually process untrusted user input, resulting in an easy and highly valuable target for attacker exploitation. Email and web browsers have enabled powerful social engineering and drive-by download attacks, which often result in the installation of malicious software. Semi-honest software can violate user expectations of security and privacy by performing unwanted actions without the user's knowledge. It is clear that a solution is needed to mitigate these threats—for this, we turn to process confinement. Unfortunately, process confinement is not yet a solved problem [17], and so the exploration of new solutions is necessary.

3 The Status Quo in Process Confinement

To understand the need for a move towards adaptive process confinement, we first need to contextualize the existing process confinement literature from an adaptive security perspective. To that end, this section presents an overview of the current process confinement landscape and discusses many of the trade-offs and drawbacks present in existing approaches. I place a particular emphasis on critiquing these systems using the adaptive process confinement definition outlined in Section 1.1.

3.1 Low-Level Techniques: The Building Blocks of Confinement

Here, I discuss many of the low-level techniques used to implement process confinement. Notably, many of these techniques were not designed with the express purpose of process confinement; rather, they are often used in *combination* by higher-level process confinement mechanisms, many of which are discussed in Section 3.2.

Discretionary Access Control Discretionary access control (DAC) forms the most basic access control mechanism in many operating systems, including popular commodity operating systems such as Linux, macOS, and Windows. First formalized in the 1983 Department of Defense standard [54], a discretionary access control system partitions system objects (e.g. files) by their respective owners, and allows resource owners to grant access to other users at their discretion. Typically, systems implementing discretionary access control also provide a special user or role with the power

to override discretionary access controls, such as the superuser (i.e. `root`) in Unix-like operating systems and the Administrator role in Windows.

While discretionary access controls themselves are insufficient to implement proper process confinement, they do form the basis for the bare minimum level of protection available on many operating systems; they are, therefore, an important part of the process confinement discussion. In many cases, user-centric discretionary access controls are abused to create per-application "users" and "groups". For instance, a common pattern in Unix-like systems such as Linux, macOS, FreeBSD, and OpenBSD is to have specific users reserved for security-sensitive applications such as network-facing daemons. The Android mobile operating system takes this one step further, instead assigning an application- or developer-specific UID (user ID) and GID (group ID) to *each* application installed on the device [28].

In theory, these abuses of the DAC model would help mitigate the potential damage that a compromised application can do to the resources that belong to other users and applications on the system. However, due to the discretionary nature of DAC, nothing is preventing a given user from simply granting permissions to all other users on the system. Further, the inclusion of non-human users into a user-centric permission model may result in a disparity between an end-user's expectations and the reality of what a "user" actually is. This gap in understanding could result in further usability and security concerns.

POSIX Capabilities Related to discretionary access control are POSIX capabilities [14, 15, 35], which can be used to grant additional privileges to specific processes, overriding existing discretionary permissions. Further, a privileged process may *drop* specific capabilities that it no longer needs, retaining those which it does need. Consequently, POSIX capabilities provide a finer-grained alternative to the all-or-nothing superuser privileges required by certain applications. For instance, a web-facing process that requires access to privileged ports has no business overriding file permissions. POSIX capabilities provide an interface for making such distinctions. Despite these benefits, POSIX capabilities have been criticized for adding additional complexity to an increasingly complex Linux permission model [14, 15]. Further, POSIX capabilities do nothing to confine processes beyond the original DAC model—rather, they help to solve the problem of overprivileged processes by limiting the privileges that they require in the first place.

Namespaces and Cgroups In Linux, *namespaces* and *cgroups* (short for control groups) allow for further confinement of processes by restricting the system resources that a process or group of processes is allowed to access. Namespaces isolate access by providing a process group a private, virtualized naming of a class of resources, such as process IDs, filesystem mountpoints, and user IDs. As of version 5.6, Linux supports eight distinct namespaces, depicted in Table 3.1. Complementary to namespaces, cgroups place limits on the use of *quantities* of system resources, such as CPU, memory, and block device I/O. Namespaces and cgroups provide fine granularity for limiting the resources that a process or process group can access. However, these are low-level mechanisms

designed to be used by application developers and higher-level frameworks and do not constitute an adaptive process confinement mechanism by themselves.

Table 3.1: Linux namespaces as of kernel version 5.6 and what they can be used to isolate.

Namespace	Isolates
PID	Process IDs (PIDs)
Mount	Filesystem mountpoints
Network	Networking stack
UTS	Host and domain names
IPC	Inter-process communication mechanisms
User	User IDs (UIDs) and group IDs (GIDs)
Time	System time
Cgroup	Visibility of cgroup membership

System Call Interposition System call interposition has historically been a prevalent process confinement technique. Several frameworks exist today for system call interposition on various Unix-like operating systems [2, 39, 40, 57]. Since system calls are used to request services from the operating system, they define the interface with the operating system's reference monitor [1]. This makes system call interposition a particularly attractive technique for the implementation of fine-grained policy enforcement mechanisms.

One of the earliest forays into system call interposition for process confinement was TRON [8]. Implemented in 1995 for the Unix operating system, TRON provided a kernelspace mechanism for enforcing *protection domains* on userspace processes. A TRON protection domain consists of its confined processes, a set of allowed operations, and a *violation handler*, which TRON invokes on policy violations. Processes configure protection domains and then employ a special `tron_fork` system call to spawn a confined child process. While TRON is not application transparent by itself, it does come with a set of userspace tools to abstract away the configuration of protection domains. Unfortunately, even with these higher-level userspace tools, TRON still assumes a certain degree of security expertise for a user to confine their applications properly.

Perhaps the most pervasive framework for interposing on system calls is ptrace [40], a process tracing and debugging framework that comes enabled in some form or another on all Unix-like operating systems. While ptrace itself is *not* designed process confinement, some research prototypes [26, 55] have leveraged it in the past. Unfortunately, ptrace is not generally considered production-safe due to its high overhead and buggy interactions with more complex programs such as sendmail, which is especially problematic considering that these are the types of programs that we often wish to confine.

Janus [26, 55] was an early exploration of process confinement using Solaris' version of ptrace. In Solaris, ptrace provides a library call interface into the procfs virtual file system and allows tracer applications to make filtering decisions on behalf of traced processes while interposing on system

calls. Janus was later ported to Linux using a modified version of Linux's `ptrace(2)` system call [55]. In Janus, a supervisor process reads a policy file and attaches itself to a confined process with ptrace. From there, security-sensitive system calls in the confined application are forwarded to the Janus supervisor process to make a policy decision. However, this approach adds considerable overhead to confined processes because ptrace requires *multiple* context switches between userspace and kernelspace to coordinate between the tracer and tracee.

To implement its policy language, Janus defines higher-level interfaces into various groups of system calls, called *policy modules*. These policy modules can filter groups of related system calls by parameterizing them with allowed actions and system objects. While this abstraction is helpful to group system calls by their related functionality, it does little to help Janus' usability, which is still tightly coupled with the underlying system calls. This lack of usability makes it difficult for a non-expert user to write an effective Janus policy.

Anderson published a study in the FreeBSD journal [2] comparing three system call interposition frameworks for three distinct Unix-like operating systems: Linux's seccomp-bpf [21, 36], OpenBSD's pledge [39], and FreeBSD's Capsicum [56, 57]. While these three frameworks all interpose on system calls, they do so with varying degrees of security, complexity, and granularity [2], so each merits study in its own regard. Figure 3.1 presents an overview of the security and granularity trade-offs in each framework.

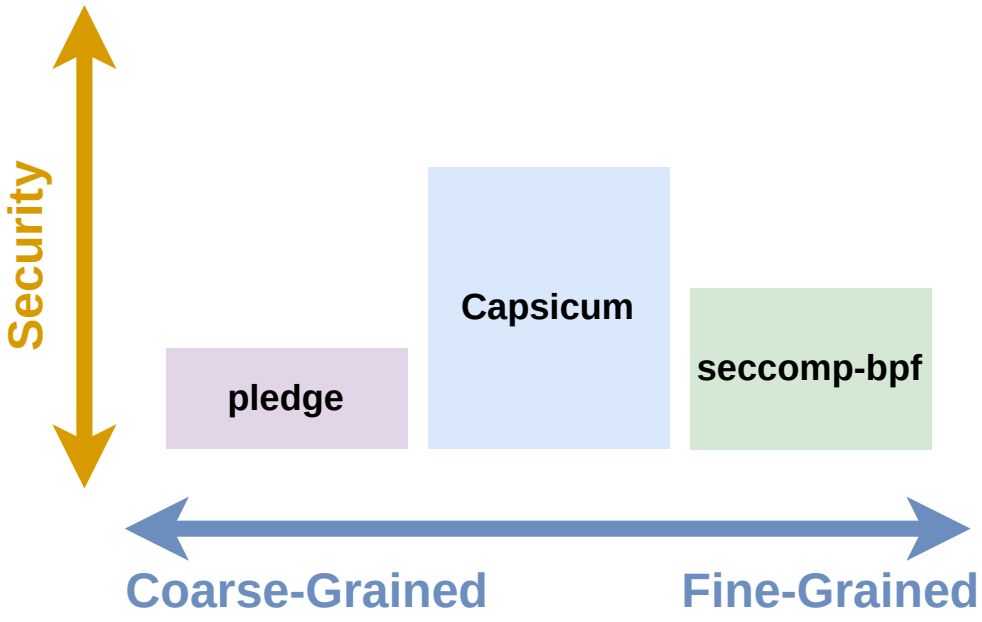

Figure 3.1: Security and granularity trade-offs of pledge, Capsicum, and seccomp-bpf.

In the original Linux seccomp implementation, processes use a special `seccomp(2)` system call to enter a secure computing state. By default, processes that have entered this state are restricted to performing `read(2)`, `write(2)`, `sigreturn(2)`, and `exit(2)` system calls. Pragmatically, this means that a process could read and write on its open file descriptors, return from invoked signal

handlers, and terminate itself. All violations of this policy would result in forced termination. In a 2012 RFC [21], Drewry introduced an extension to seccomp enabling the use of BPF programs[2] for the defining filters on system call arguments. This extension, dubbed seccomp-bpf, enables creating fine-grained seccomp policies that filter on system call numbers and arguments, providing a high degree of control to applications that wish to sandbox themselves.

Despite the high degree of control that seccomp-bpf offers to applications, it has severe usability and security concerns, rendering it an unacceptable solution for ad-hoc confinement by end-users. Classic BPF [38] is a rather arcane bytecode language, and writing classic BPF programs by hand is a task left only to expert users. Further, seccomp-bpf policy is easy to misconfigure, resulting in potential security violations; for instance, an attacker may entirely circumvent a policy that specifies restrictions on the `open(2)` system call but not `openat(2)`. Finally, despite userspace library efforts to abstract away the underlying BPF programs [45], seccomp-bpf remains accessible only to application developers with significant security expertise.

OpenBSD's pledge [39] takes a simpler, coarser-grained approach to system call filtering than seccomp-bpf, instead grouping system calls into high-level semantically meaningful categories, such as `stdio` which includes `read(2)` and `write(2)`, for example [2]. This coarse granularity and simplicity provide increased usability but come at the expense of expressiveness. There is no canonical way to distinguish subsets of system call groups or filter system calls by their arguments. Despite its increased usability for developers, pledge still suffers from a lack of application transparency just as seccomp-bpf does, meaning that it is only suitable for application developers rather than end-users.

Unlike seccomp-bpf and pledge, which apply filtering rules to system calls directly, FreeBSD's Capsicum takes the approach of restricting access to global namespaces via a capability-based implementation [57]. In Capsicum, a process enters *capability mode* using a special `cap_enter` system call. Once in capability mode, access to global namespaces is restricted to the capabilities requested by the process. These capabilities are inherited across `fork(2)` and `execve(2)` calls. Much like seccomp-bpf and pledge, however, Capsicum is *not* application transparent and is designed for use by developers rather than end-users.

Linux Security Modules The Linux Security Modules (LSM) API [58] provides an extensible security framework for the Linux kernel, allowing for the implementation of powerful kernelspace security mechanisms that can be chained together. LSM works by integrating a series of strategically placed *security hooks* into kernelspace code. These hooks roughly correspond with boundaries for the modification of kernel objects. Multiple security implementations can hook into these LSM hooks and provide callbacks that generate audit logs and make policy decisions. Figure 3.2 depicts the LSM architecture in detail.

The LSM API sits at a level of abstraction just above the system call API—a single LSM hook may cover multiple system calls and a single system call may contain multiple such LSM hooks. For

[2]BPF is a special bytecode language, originally implemented for packet filtering in BSD Unix [38]. seccomp-bpf, retrofits BPF for the purpose of filtering system calls instead.

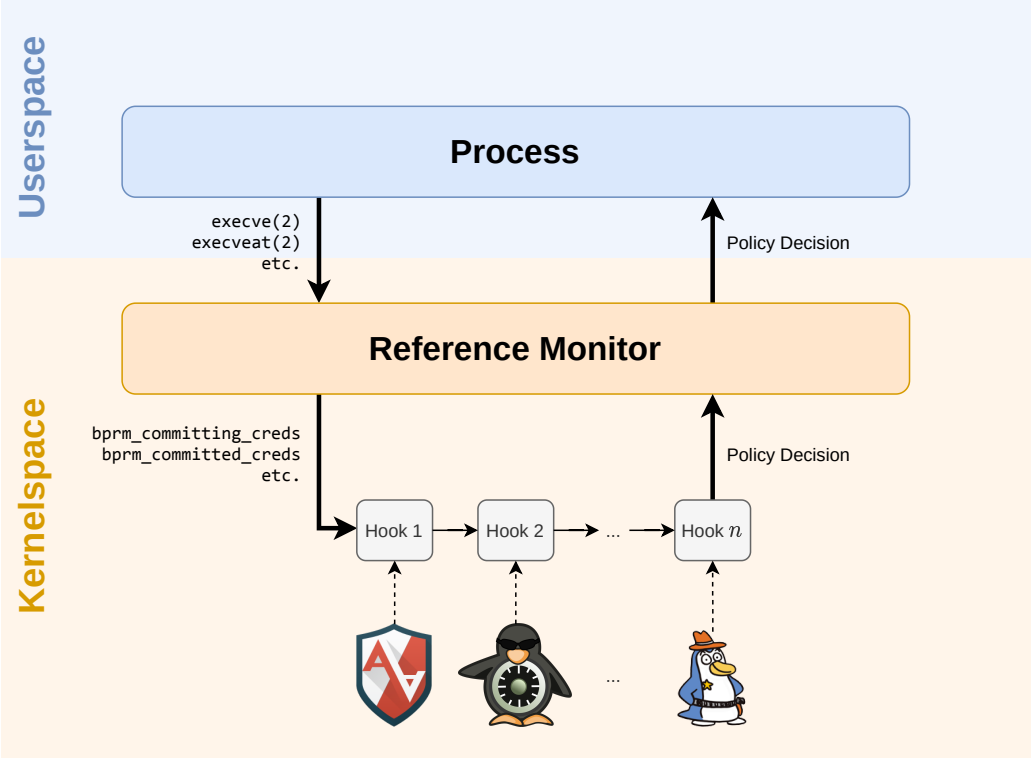

Figure 3.2: The LSM architecture. Note the many-to-many relation between access requests and hook invocations. Multiple LSM hooks may be chained together, incorporating policy from many security mechanisms. All hooks must agree to allow the access or it will be denied.

instance, the `execve(2)` and `execveat(2)` calls both result in a call to the `bprm_committing_creds` and `bprm_committed_creds` hooks. This provides a nice level of abstraction compared to system-call-based approaches like seccomp-bpf [21, 36] in that a single LSM hook can cover all closely related security events (recall the issue of `open(2)` vs `openat(2)` in seccomp-bpf).

The Linux kernel contains several in-tree LSM-based security modules, which may be enabled by default on certain distributions. Many such modules implement *mandatory access control* (MAC) schemes, which enable fine-grained access control that can limit the privileges of *all users*—even the superuser. SELinux [47] and AppArmor [16] are two such MAC LSMs, each with its own policy semantics. I discuss each in turn.

SELinux [47] was originally developed by the NSA as a Linux implementation of the Flask [52] security model. Under SELinux, system subjects (users, processes, etc.) and system objects (files, network sockets, etc.) are each assigned corresponding labels. Security policy is then written based on these labels, specifying the allowed access patterns between a particular object type and subject type. SELinux's policy language is famously arcane [44], and despite multiple efforts to introduce automated policy generation [37, 48, 50], writing and auditing SELinux security policy remains a task for security experts rather than end-users. Further, due to the difficulty of writing and auditing the complex SELinux policy language, there is a natural tendency for human policy authors to err on the side of over-permission, violating the principle of least privilege.

AppArmor (originally called SubDomain) [16] is often touted as a more usable alternative to SELinux, although usability studies have shown that this claim merits scrutiny [44]. Rather than basing security policy on labelling system subjects and objects, AppArmor instead employs path-based enforcement. AppArmor defines policy in per-application profiles, which contain rules specifying what system objects the application can access. System objects are identified directly (for example, via pathnames, socket classes, or IP network addresses) rather than labelled. AppArmor also supports the notion of *changing hats*, wherein a process may change its AppArmor profile under certain conditions specified in the policy. Although AppArmor profiles are more conforming to standard Unix semantics than their SELinux counterparts, users who wish to write AppArmor policy still require a considerable amount of knowledge about operating system security [44].

3.2 High-Level Approaches

In this section, I examine higher-level approaches to process confinement; these are approaches that typically combine more than one of the lower-level techniques discussed previously. Although these solutions usually improve usability by providing higher-level policy abstractions, they are not strictly any better than their lower-level counterparts. Their policy is often coarse-grained, overly-permissive, and difficult to audit.

Containerized Package Management In Linux, a recent trend of *containerized package management* has emerged, allowing users to download applications as packages which are then confined using a combination of lower-level techniques exposed by the operating system. Among the most popular of these containerized package management frameworks are Docker [20], Snap [49], and Flatpak [24]. Typically, a package maintainer or application author would write a high-level package manifest declaring the privileges required by the application. The package manager would then implicitly translate this policy into lower-level policies for the underlying process confinement mechanisms (such as those presented in Section 3.1). Figure 3.3 depicts this architecture in detail.

Despite their widespread adoption, these containerized package management solutions are far from perfect [53]. Due to the coarse granularity of package manifests and the high complexity of the underlying policy enforcement mechanisms, policy written for these containerized package managers tends towards gross over-permission. Auditability of the policy also suffers due to the disparity between user *expectations* of protection described in package manifests and the *actual policy* enforced in practice. In effect, five or six lines of high-level policy in a package manifest may end up generating thousands of lines of complex, low-level policy across multiple underlying process confinement mechanisms; thus, auditing the generated security policy also becomes a seemingly hopeless task, even for seasoned security experts.

As a motivating example, consider the package manifest for Snap's Nextcloud[3] package, which includes the Apache `httpd` webserver. The package manifest lists the following permissions for

[3] Available: https://snapcraft.io/nextcloud

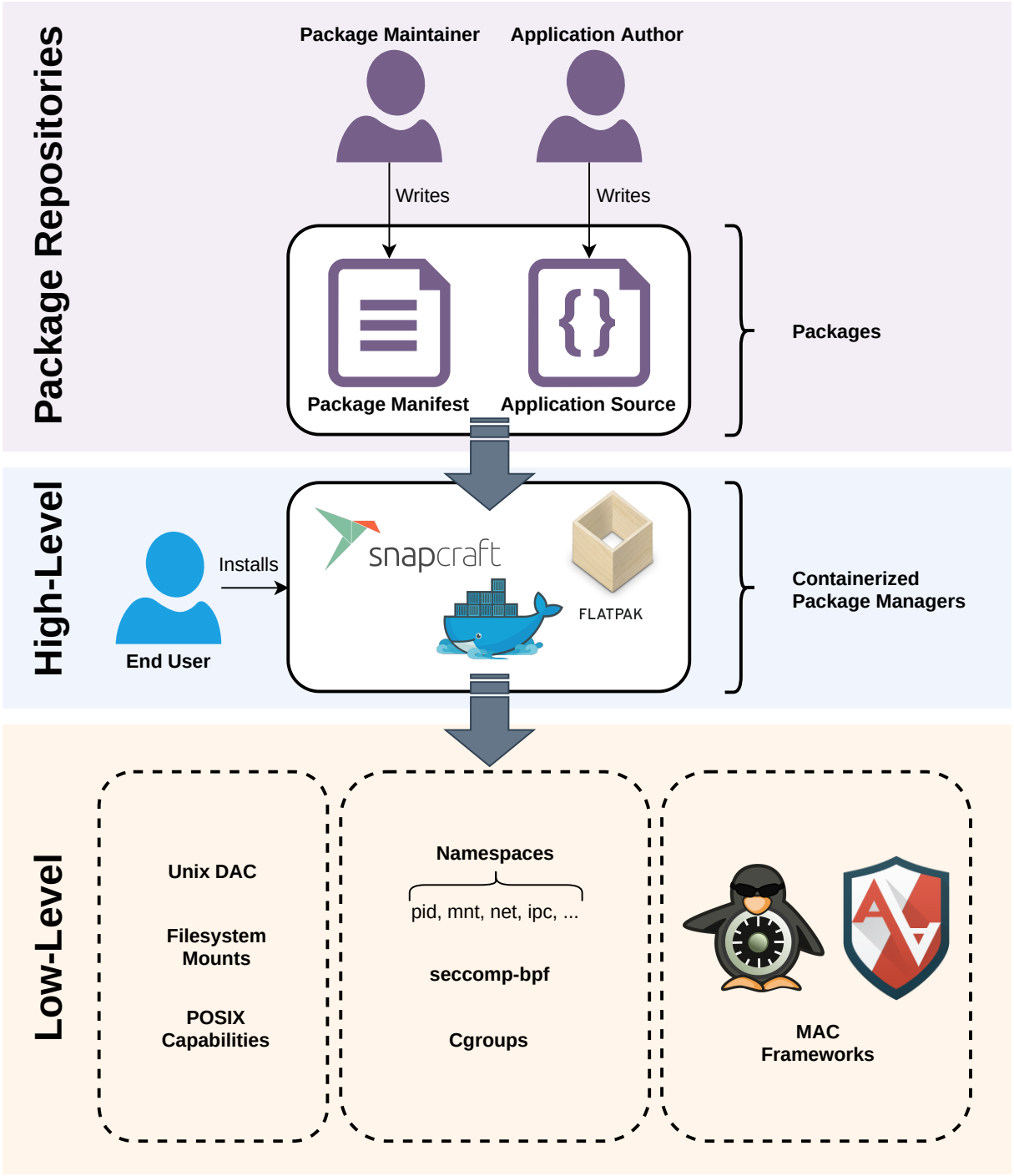

Figure 3.3: The basic architecture of containerized package management solutions for Linux, such as Snapcraft [49], Flatpak [24], and Docker [20]. Package maintainers write high-level, coarse-grained package manifests, which are then compiled into policy for lower-level process confinement mechanisms to enforce.

Apache `httpd`:

```
confine: strict
/* ... */
apache:
  daemon: simple
  plugs:
  - network
  - network-bind
  - removable-media
```

The result, after policy generation, is 411 seccomp-bpf filters and a 653 line AppArmor policy. The generated policy is overly permissive and covers a broad scope of capabilities—far beyond what would be expected based on the three lines of security policy defined in the manifest. For instance, the AppArmor profile allows the execution of over 120 standard shell utilities, none of which are indicated in the package manifest. There are also frequent mismatches between the generated seccomp-bpf and AppArmor policy files. For example, many socket operations denied in the AppArmor policy are allowed in the seccomp-bpf policy. This mismatch adds an extra layer of difficulty to auditing the generated policy.

Ultimately, the problem with containers as an isolation mechanism lies in the complexity of the underlying process confinement mechanisms that they leverage. Namespaces, cgroups, and filesystem mounts virtualize the containers, while complex systems like seccomp, SELinux, and AppArmor enforce least privilege. The simplicity of high-level package manifests belies the complexity underneath, wherein full userlands must be secured for each application. The resulting simplification can obfuscate overly-permissive policies, resulting in a false sense of security, particularly when the underlying process confinement mechanisms are misconfigured transparently to the end-user.

Confinement with Wrapper Sandboxes Separate but related to containerized application management are *wrapper sandboxes*. Like containers, these wrappers use a combination of multiple lower-level process confinement techniques to achieve confinement. The user typically executes a wrapper application that sets up the sandbox before launching the target application (e.g. `generic-wrapper-application firefox`). Some examples of these wrappers include Firejail [23], Bubblewrap [9], and Mbox [33].

Even very early on in the development of Unix process confinement mechanisms, it was recognized that wrapper confinement applications could provide significant usability benefits by abstracting away the underlying details of process confinement. For instance, TRON [8] provided a wrapper application for its protection-domain-based enforcement back in 1995. While these wrapper applications often come with usability and transparency benefits, they suffer from auditability and robustness issues not dissimilar from the containerized package managers discussed above.

Firejail [23] uses seccomp-bpf and Linux namespaces to confine applications. Recent versions also include a generic AppArmor profile to provide an extra layer of basic protection. As with

the other container-based approaches we have seen thus far, policy is defined using per-application profiles, which are then compiled into the underlying policy mechanisms. While these policy files are quite high-level and designed to be used out the box (on Arch Linux, the latest version of Firejail comes with over 1000 pre-configured profiles), their auditability and robustness suffer due to the coarse-granularity of rules. For instance, Firejail takes an all-or-nothing approach to file access, specified using either `whitelist`, `blacklist`, or `noblacklist`. Further, it is possible to combine all three of these keywords on a shared pathname, leading to results that may be unpredictable for end-users[4].

Kim and Zeldovich take a rather different approach with Mbox [33]; instead of using policy profiles, access in Mbox is specified using coarse-grained flags passed to a wrapper application. To confine applications, Mbox leverages a combination of seccomp-bpf filters and ptrace to interpose on system calls and re-write their arguments. Applications are confined to a sandboxed filesystem by re-writing the arguments to the `open(2)`, `read(2)`, and `write(2)` system calls. Since Mbox policy is expressed using command line arguments, auditing Mbox is impossible without actually reading its source code. Further, a user requires sufficient knowledge to know which command line arguments should be used with which application.

Like Mbox, Bubblewrap [9] specifies confinement rules for applications using command line arguments. Bubblewrap confines applications using a combination of filesystem mounts, Linux namespaces and seccomp-bpf rules, and is designed to work cooperatively with other container-based approaches like Flatpak [24]. Just as with Mbox, the auditability and usability of Bubblewrap suffers due to the advanced knowledge required to run an application using the appropriate confinement arguments, which may end up being overly permissive regardless.

3.3 (Semi-)Automated Policy Generation

Usability and transparency issues represent a common theme among several of the process confinement mechanisms I have discussed thus far. This is particularly true for highly complex, low-level mechanisms like SELinux [47], AppArmor [16], and seccomp-bpf [21, 36]. It is tempting to turn toward the notion of *automated policy generation* to alleviate these issues.

Both SELinux and AppArmor include mechanisms for automated and semi-automated policy generation. AppArmor's `aa-easyprof` [3] guides the user through policy generation, and supplementary templates and abstractions can be provided to bootstrap the policy generation process. However, policy generated using this method tends toward over-permission, and its quality is highly dependent on the quality of the provided policy templates [3]. AppArmor also supports two lower-level policy generation tools. `aa-genprof` [4] generates policy using AppArmor's audit logs, while `aa-logprof` [5] does the same thing with guided input from the user. Both these approaches have the advantage of higher transparency to the end-user but may generate a more restrictive policy

[4]See this GitHub issue for an example: https://github.com/netblue30/firejail/issues/1569

that could require manual modification to function.

SELinux policy generation was originally supported via the `audit2allow` [48] command line tool. This tool functions similarly to `aa-genprof` [4] above, automatically generating policy with the help of SELinux's audit logs. Sniffen *et al.*recognized the need for improving upon the existing policy generation in SELinux and introduced a guided policy generation approach, `polgen` [50]. `polgen` semi-automates the policy generation process using modified version of the `strace`[5] command line utility, which interposes on system calls and analyzes patterns in their arguments to infer policy. Policy is expressed in an intermediate representation language called PSL, which is then fed into a modelling program that constructs an information flow graph and a type generation program to generate new labels for system objects. In the end, the resulting PSL policy is compiled into the actual SELinux policy to be installed on the system. Unfortunately, this approach does little to alleviate SELinux's primary usability concerns, as the generated policy is still tied down to SELinux's arcane labelling and type enforcement mechanisms.

The 2006 introduction of the SELinux reference policy [41] provided re-usable modules that could be plugged into new policy files using high-level interfaces. This motivated MacMillan to develop Madison [37], yet another take on SELinux policy generation. Rather than re-inventing the wheel, Madison was designed to be complementary to existing policy generation mechanisms. The idea was that Madison would be used to generate the *reference policy* itself, which could then be integrated with new policy files using existing tools like `polgen` or by hand. Miranda's approach does help to abstract away the underlying SELinux implementation details but ultimately falls short of its goal of improving usability. Users still need to understand the underlying SELinux policy language if they are to have any hope of auditing or modifying the generated reference policy files.

While SELinux and AppArmor included policy generation as an afterthought, other policy generation mechanisms have taken an alternative approach, building the initial process confinement mechanism with policy generation in mind. Systrace [42] builds policy by observing individual system calls on the running system along with their relevant arguments. Policy may be built automatically or semi-automatically through pop-up dialogs on mediated security events. Systrace's policy language design is reminiscent of the higher level seccomp-bpf policy representations found in frameworks like Snap [49]. Although Systrace's policy language design integrates nicely with its policy generation mechanism, its usability is impacted by the fact that the end-user must be familiar with system call semantics to read, understand, and modify Systrace policy.

Inoue and Forrest [31, 32] recognized an opportunity to apply a similar technique to the Java runtime's extant sandboxing mechanisms [27]. Inoue and Forrest's work enables the dynamic creation of Java security policy by instrumenting Java's `checkPermission` method and recording cases where permission would be denied under normal operation. Java sandboxing is a unique case in that it is not strictly designed to be written or audited by the end-user; rather, it is the application programmers that write policy for their own applications. Dynamically generating a minimal security

[5] `strace` uses the `ptrace(2)` system call under the hood.

policy for a Java application could have great benefits for Java developers who could then distribute this policy with their applications transparently to the end-user. However, an ideal security policy should be understandable by *all* stakeholders—this approach makes no effort to move away from the complex semantics of Java security policy; rather, it tries its best to make them as transparent as possible.

While policy generation can indeed help with transparency issues in policy authorship, it does not necessarily fully alleviate usability concerns. So long as the underlying confinement language remains complex, it will be difficult for a user to audit the generated policy. Manual modification of the generated policy also remains a challenge here; it would be desirable to make this easy so as to enable ad hoc process confinement by end-users and painless debugging of overly restrictive auto-generated policies. It is clear that policy generation represents at least a partial step towards adaptive process confinement, but it does not completely close the gap.

3.4 (Semi-)Automated Policy Audit

To attempt to solve the problem of auditing complex policy languages (e.g. SELinux [47] and AppArmor [16]), many researchers have turned to the same automation techniques used in the policy generation literature. Chen *et al.*wrote VulSAN [10] to alleviate some of the auditability concerns present in SELinux and AppArmor and to compare the default protection offered by both systems. VulSAN builds a Prolog[6] database of facts about the MAC policy (i.e. the SELinux or AppArmor policy), Unix DAC, and state of the running system. Users then issue queries encoding a threat model—an attack goal and a set of assumptions about an attacker's resources—and VulSAN generates an *attack graph* describing the shortest path of exploitation needed to realize the attack.

While VulSAN can certainly help security experts analyze the protection benefits afforded by their choice of MAC implementation, it offers little benefit to users who are not already well-versed in operating system security (precisely the same skillset one would need to audit SELinux and AppArmor policy in the first place). Further, VulSAN is capable of identifying vulnerabilities in security policy but incapable of describing how to fix these vulnerabilities.

iOracle [18] takes a similar approach to VulSAN but applies it to security policy under the iOS mobile operating system instead. Unlike the majority of the confinement mechanisms I have discussed thus far, iOS's sandbox is proprietary, closed-source software. This makes auditing such a system a particularly daunting task without any knowledge about the inner workings of the system. To rectify this lack of auditability, iOracle employs reverse engineering and static analysis techniques to infer security policy from compiled policy binaries. It then builds out a similar logic database to that of VulSAN [10], allowing users to issue queries about the security properties of applications on the system. Unlike VulSAN, iOracle attempts to describe how identified vulnerabilities might be fixed. However, iOracle still assumes considerable security knowledge on the part of the user, as

[6]Prolog is a logic programming language wherein a developer writes a series of *facts*, which can then be used to resolve *queries*. In this respect, Prolog programs may be thought of as a sort of *logic database*.

the user needs to formulate correct queries to make use of the model.

Although systems like VulSAN and iOracle can help defenders by automating the audit of security policy, they assume that defenders have enough security knowledge to issue the correct security queries. After all, even a perfect oracle is completely useless if one does not know the correct questions to ask. These systems do little to help an average user develop and audit their own security policy, and so they cannot be considered truly adaptive security mechanisms in their own right.

3.5 Taint Propagation

Unlike traditional process confinement mechanisms, which seek to limit the set of possible actions that a program can ever take, the notion of *taint propagation* directly seeks to limit the impact that untrusted user input can have on the running system. This is typically done by marking data with a special taint flag and propagating this flag across function calls; according to policy, using tainted data to perform a specific operation would cause that operation to fail. Perl's "taint mode" [30] was the first implementation of such a tainting mechanism on program data. A special command line flag passed to a Perl program triggers it to run in taint mode, wherein untrusted input from the user cannot be directly passed into a pre-determined list of unsafe built-in functions. To get around this restriction, the developer must implement sanity checks on the data, which would then *untaint* it. While Perl's taint mode provides some inherent level of protection, it mostly exists as a means to encourage developers to implement proper sanity checks on untrusted data, and incorrect sanity checks may result in no protection at all [30]. Beyond Perl, similar taint mechanisms have since been implemented for a myriad of other interpreted programming languages[7].

Conti, Bello, and Russo [7, 12] implemented a novel tainting mechanism for Python as a library, and argue that it can be used to effectively confine modern web applications in the presence of untrusted user data. By implementing tainting at the language level rather than at the level of the interpreter itself, the authors argue that the approach can be easily adapted to accommodate new use cases and data types [7, 12]. This approach is in stark contrast to the original taint implementation in Perl, which focuses on securing only certain access primitives [30].

Findlay, Somayaji, and Barrera's bpfbox [22] approaches tainting from a different angle. Instead of tainting data, bpfbox policy defines *taint rules*, specifying access patterns that should taint the *entire process*. bpfbox does not enforce any security policy until this tainting occurs. By recognizing when a process has entered an insecure state, bpfbox can greatly simplify its security policy to focus on restricting only a subset of a program's functionality.

[7] To name a few: Ruby, PHP, and Python now all provide native tainting support [12].

4 Towards Truly Adaptive Process Confinement

Throughout this literature review, I have examined several process confinement implementations which span a variety of techniques (both high-level and low-level), operating systems, and programming languages. Despite a rich history of process confinement dating back nearly five decades [34], none of the solutions presented here constitutes a full solution to the process confinement problem.

Complex mandatory access control mechanisms like SELinux [47] and AppArmor [16] provide incredibly precise policy languages, but are only accessible to knowledgeable security experts; higher-level mechanisms like Docker, Snap, and Flatpak offer only overly-permissive, difficult-to-audit package manifests, and cobble together a myriad of mostly unrelated solutions to implement their confinement under the hood.

Automated policy generation and audit mechanisms tend to be too tied down to existing enforcement mechanisms and operating system semantics to be truly useful to end-users; a user without significant security expertise cannot hope to understand their output. Taint propagation mechanisms suffer a similar fate, as they are designed to be used by application developers rather than end-users, and buggy applications that conform to these taint mechanisms can still expose the system to risk.

To truly solve the problem of process confinement, I propose that we need to move away from the traditional process confinement mechanisms discussed above, and instead move towards *truly adaptive* process confinement mechanisms. A truly adaptive process confinement would help solve the process confinement problem by providing defenders with tools that can automate the protection of a running system without relying on complex policy languages or advanced security knowledge. These mechanisms would be mostly transparent to the end user, easy to use and reconfigure, and effective at responding to novel attacks.

4.1 Bridging the Gap with Anomaly Detection

In anomaly detection, the behaviour of the running system is compared with previously observed behaviour to determine if any deviations have occurred; these deviations are flagged as anomalous behaviour, which may be correlated with an active attack [25]. Many anomaly detection systems include a response component, which can thwart detected attacks altogether, while others simply report possible attacks to an administrator.

At first glance, it may seem as though anomaly detection obviates the need for process confinement entirely. If an anomaly detection system could detect anomalous behaviour with perfect accuracy, this would be true. However, anomaly detection is not an exact science, and even the most robust anomaly detection systems are subject to false positives and false negatives [25]. In process confinement, we have a similar yet subtly different problem: policy can be overly permissive (c.f. false negatives) or too restrictive (c.f. false positives).

There are many possible reasons why policy may be imprecise, as we have seen in the process confinement literature. For instance, a policy language might be too coarse-grained to confine a process properly; on the other hand, another policy language might be too complicated for a human policy author to have any hope of understanding it. Ultimately, this is symptomatic of a fundamental trade-off in process confinement: terseness of policy versus expressiveness of policy. A terse policy will be easy to understand and write but may not be expressive enough to capture all possible behaviour, while an expressive policy may be difficult for users to write and audit.

I hypothesize that the key to solving this trade-off and creating a genuinely adaptive process confinement mechanism lies in bridging the gap between anomaly detection and process confinement. Anomaly detection techniques could be leveraged to generate the building blocks of the policy language itself. Since this policy language would be generated using behaviour observed on the running system, its semantics would conform precisely to the sort of behaviour that defenders are interested in confining. Applying meaningful abstractions to this generated policy language could allow it to be used by humans without sacrificing its underlying precision.

As a motivating example, consider observed sequences of system calls: a `fork(2)` followed by an `execve(2)` could be abstracted into one rule called `run_program`; the complex sequence of `fstat`s, `mmap`s, `access`es, `openat`s, and `read`s at the beginning of a C program could be abstracted into another rule called `load_libc`. The user doesn't need to know or care about the underlying details of these rules, yet the basis of the policy language conforms precisely with the semantics of the system.

In the existing process confinement literature, this idea of bridging the gap between anomaly detection and process confinement is not unheard-of. Provos' Systrace [42] and Inoue and Forrest's technique for dynamic sandboxing in Java [31, 32] are two excellent examples of this. Both of these mechanisms analyze running software's behaviour to generate policy automatically; however, the resulting policy is often complex since it is expressed in a confinement language that directly conforms to the underlying system's semantics. An ideal solution would achieve a generated policy that is simultaneously precise and understandable by end-users. By generating the confinement language itself, this complexity could be abstracted away while preserving the underlying semantics necessary for precision.

5 Conclusion

Process confinement is a challenging problem that has been studied for nearly five decades [34]. Many proposed solutions exist, yet none has solved the fundamental problem of policy precision versus usability. I propose that the key to solving this problem is approaching it from an adaptive point of view, borrowing techniques from the anomaly detection space to create a process confinement mechanism that infers its policy language from the running system. By developing a truly adaptive process confinement mechanism, we can enhance the ability of *all* defenders, security experts or

otherwise, to protect their running processes from exploitation by malicious parties.

References

[1] James P. Anderson, "Computer Security Technology Planning Study," Tech. Rep. ESD-TR-73-51, 1973, Section 4.1.1. [Online]. Available: https://csrc.nist.gov/csrc/media/publications/conference-paper/1998/10/08/proceedings-of-the-21st-nissc-1998/documents/early-cs-papers/ande72.pdf.

[2] James P. Anderson, "A Comparison of Unix Sandboxing Techniques," *FreeBSD Journal*, 2017. [Online]. Available: http://www.engr.mun.ca/~anderson/publications/2017/sandbox-comparison.pdf.

[3] AppArmor, *aa-easyprof(8)*, Linux user's manual. [Online]. Available: https://manpages.ubuntu.com/manpages/precise/man8/aa-easyprof.8.html.

[4] AppArmor, *aa-genprof(8)*, Linux user's manual. [Online]. Available: https://manpages.ubuntu.com/manpages/precise/man8/aa-genprof.8.html.

[5] AppArmor, *aa-logprof(8)*, Linux user's manual. [Online]. Available: https://manpages.ubuntu.com/manpages/precise/man8/aa-logprof.8.html.

[6] Lynn Erla Beegle, "Rootkits and Their Effects on Information Security," *Information Systems Security*, vol. 16, no. 3, pp. 164–176, 2007. DOI: 10.1080/10658980701402049.

[7] Luciano Bello and Alejandro Russo, "Towards a Taint Mode for Cloud Computing Web Applications," in *Proceedings of the 2012 Workshop on Programming Languages and Analysis for Security, PLAS 2012, Beijing, China, 15 June, 2012*, ACM, 2012, p. 7. DOI: 10.1145/2336717.2336724.

[8] Andrew Berman, Virgil Bourassa, and Erik Selberg, "TRON: Process-Specific File Protection for the UNIX Operating System," in *Proceedings of the USENIX 1995 Technical Conference*, The USENIX Association, 1995, pp. 165–175. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.56.9149&rep=rep1&type=pdf.

[9] Bubblewrap, *Bubblewrap*, 2020. [Online]. Available: https://github.com/containers/bubblewrap (visited on 10/25/2020).

[10] Hong Chen, Ninghui Li, and Ziqing Mao, "Analyzing and Comparing the Protection Quality of Security Enhanced Operating Systems," in *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, 2009. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2017/09/Chen.pdf.

[11] Frederick B. Cohen, "A Secure World-Wide-Web Daemon," *Comput. Secur.*, vol. 15, no. 8, pp. 707–724, 1996. DOI: 10.1016/S0167-4048(96)00009-0.

[12] Juan José Conti and Alejandro Russo, "A Taint Mode for Python via a Library," in *Information Security Technology for Applications - 15th Nordic Conference on Secure IT Systems, NordSec 2010, Espoo, Finland, October 27-29, 2010, Revised Selected Papers*, vol. 7127, Springer, 2010, pp. 210–222. DOI: 10.1007/978-3-642-27937-9_15.

[13] Fernando J. Corbató, Merwin-Daggett, given=Marjorie, giveni=M., "An Experimental Time-Sharing System," in *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*, ser. AIEE-IRE '62 (Spring), San Francisco, California: Association for Computing Machinery, 1962, pp. 335–344, ISBN: 9781450378758. DOI: 10.1145/1460833.1460871.

[14] Jonathan Corbet, "A bid to resurrect Linux capabilities," *LWN*, 2006. [Online]. Available: https://lwn.net/Articles/199004/.

[15] Jonathan Corbet, "File-based capabilities," *LWN*, 2006. [Online]. Available: https://lwn.net/Articles/211883/.

[16] Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Virgil Gligor, "SubDomain: Parsimonious Server Security," in *Proceedings of the 14st Large Installation Systems Administration Conference (LISA)*, New Orleans, LA, United States: USENIX Association, 2000. [Online]. Available: https://www.usenix.org/legacy/event/lisa2000/full_papers/cowan/cowan.pdf.

[17] Alexander Crowell, Beng Heng Ng, Earlence Fernandes, and Atul Prakash, "The Confinement Problem: 40 Years Later," *Journal of Information Processing Systems*, vol. 9, no. 2, pp. 189–204, 2013. DOI: 10.3745/JIPS.2013.9.2.189. [Online]. Available: http://jips-k.org/journals/jips/digital-library/manuscript/file/22579/JIPS-2013-9-2-189.pdf.

[18] Luke Deshotels, Razvan Deaconescu, Costin Carabas, Iulia Manda, William Enck, Mihai Chiroiu, Ninghui Li, and Ahmad-Reza Sadeghi, "iOracle: Automated Evaluation of Access Control Policies in iOS," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ser. ASIACCS '18, Incheon, Republic of Korea: Association for Computing Machinery, 2018, pp. 117–131, ISBN: 9781450355766. DOI: 10.1145/3196494.3196527.

[19] Discord, *Discord Privacy Policy*. [Online]. Available: https://discord.com/privacy (visited on 10/25/2020).

[20] Docker, *Docker Security*, 2020. [Online]. Available: https://docs.docker.com/engine/security/security (visited on 10/25/2020).

[21] Will Drewry, "Dynamic seccomp policies (using BPF filters)," Internet RFC, 2012. [Online]. Available: https://lwn.net/Articles/475019/.

[22] William Findlay, Anil Somayaji, and David Barrera, "bpfbox: Simple Precise Process Confinement with eBPF," in *Proceedings of the 2020 ACM Cloud Computing Security Workshop (CCSW'2020)*, To appear, Nov. 2020. DOI: 10.1145/3411495.3421358. [Online]. Available: https://www.cisl.carleton.ca/~will/written/conference/bpfbox-ccsw2020.pdf.

[23] Firejail, *Firejail*, 2020. [Online]. Available: https://firejail.wordpress.com (visited on 10/25/2020).

[24] Flatpak, *Sandbox Permissions*, 2020. [Online]. Available: https://docs.flatpak.org/en/latest/sandbox-permissions.html (visited on 10/25/2020).

[25] Stephanie Forrest, Steven A. Hofmeyr, and Anil Somayaji, "The Evolution of System-Call Monitoring," in *Twenty-Fourth Annual Computer Security Applications Conference, ACSAC 2008, Anaheim, California, USA, 8-12 December 2008*, IEEE Computer Society, 2008, pp. 418–430. DOI: 10.1109/ACSAC.2008.54.

[26] Ian Goldberg, David Wagner, Randi Thomas, and Eric Brewer, "A Secure Environment for Untrusted Helper Applications (Confining the Wily Hacker)," in *Proceedings of the Sixth USENIX UNIX Security Symposium*, The USENIX Association, 1996. [Online]. Available: https://www.usenix.org/legacy/publications/library/proceedings/sec96/full_papers/goldberg/goldberg.pdf.

[27] Li Gong, Gary Ellison, and Mary Dageforde, "Inside Java 2 Platform Security: Architecture, API Design, and Implementation," 1993, Second Edition.

[28] Google, *Android Security Features*, Android security documentation. [Online]. Available: https://source.android.com/security/features (visited on 10/26/2020).

[29] Robert M. Graham, "Protection in an information processing utility," *Communications of the ACM*, vol. 11, no. 5, pp. 365–369, 1968, ISSN: 0001-0782. DOI: 10.1145/363095.363146.

[30] Andrew Hurst, *Analysis of Perl's Taint Mode*, 2004. [Online]. Available: http://hurstdog.org/papers/hurst04taint.pdf.

[31] Hajime Inoue, "Anomaly detection in dynamic execution environments," Ph.D. dissertation, University of New Mexico, 2005. [Online]. Available: https://www.cs.unm.edu/~forrest/dissertations/inoue-dissertation.pdf.

[32] Hajime Inoue and Stephanie Forrest, "Inferring Java Security Policies through Dynamic Sandboxing," in *International Conference on Programming Languages and Compilers (PLC'05)*, 2005. [Online]. Available: https://www.cs.unm.edu/~forrest/publications/inoue-plc-05.pdf.

[33] Taesoo Kim and Nickolai Zeldovich, "Practical and Effective Sandboxing for Non-root Users," in *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, USENIX Association, 2013, pp. 139–144. [Online]. Available: https://www.usenix.org/conference/atc13/technical-sessions/presentation/kim.

[34] Butler W. Lampson, "A Note on the Confinement Problem," *Communications of the ACM*, vol. 16, no. 10, pp. 613–615, 1973, ISSN: 0001-0782. DOI: 10.1145/362375.362389.

[35] Linux, *capabilities(7)*, Linux user's manual. [Online]. Available: https://linux.die.net/man/7/capabilities.

[36] Linux, *Seccomp BPF (SECure COMPuting with filters)*, Linux kernel documentation. [Online]. Available: https://static.lwn.net/kerneldoc/userspace-api/seccomp_filter.html (visited on 10/27/2020).

[37] Karl MacMillan, "Madison: A new approach to policy generation," in *SELinux Symposium*, 2007. [Online]. Available: http://selinuxsymposium.org/2007/papers/08-polgen.pdf.

[38] Steven McCanne and Van Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," *USENIX Winter*, vol. 93, 1992. [Online]. Available: https://www.tcpdump.org/papers/bpf-usenix93.pdf.

[39] OpenBSD, *pledge(2)*, OpenBSD user's manual. [Online]. Available: https://man.openbsd.org/pledge.2.

[40] Pradeep Padala, "Playing with ptrace, Part I," *Linux Journal*, vol. 2002, no. 103, p. 5, 2002. [Online]. Available: https://www.linuxjournal.com/article/6100.

[41] Christopher J. PeBenito, Frank Mayer, and Karl MacMillan, "Reference Policy for Security Enhanced Linux," in *SELinux Symposium*, 2006. [Online]. Available: http://selinuxsymposium.org/2006/papers/05-refpol.pdf.

[42] Niels Provos, "Improving Host Security with System Call Policies," in *Proceedings of the 13th USENIX UNIX Security Symposium*, The USENIX Association, 2003. [Online]. Available: http://citi.umich.edu/u/provos/papers/systrace.pdf.

[43] Dennis M. Ritchie and Ken Thompson, "The UNIX Time-Sharing System," in *Proceedings of the Fourth ACM Symposium on Operating System Principles*, ser. SOSP '73, New York, NY, USA: Association for Computing Machinery, 1973, p. 27, ISBN: 9781450373746. DOI: 10.1145/800009.808045.

[44] Z. Cliffe Schreuders, Tanya Jane McGill, and Christian Payne, "Towards Usable Application-Oriented Access Controls," in *International Journal of Information Security and Privacy*, vol. 6, 2012, pp. 57–76. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.963.860&rep=rep1&type=pdf.

[45] Seccomp Contributors, *libseccomp*. [Online]. Available: https://github.com/seccomp/libseccomp (visited on 10/27/2020).

[46] Hovav Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07, Alexandria, Virginia, USA: Association for Computing Machinery, 2007, pp. 552–561, ISBN: 9781595937032. DOI: 10.1145/1315245.1315313. [Online]. Available: https://doi-org.proxy.library.carleton.ca/10.1145/1315245.1315313.

[47] Stephen Smalley, Chris Vance, and Wayne Salamon, "Implementing SELinux as a Linux security module," 43, vol. 1, 2001, p. 139. [Online]. Available: https://www.cs.unibo.it/~sacerdot/doc/so/slm/selinux-module.pdf.

[48] Justin R. Smith, Yuichi Nakamura, and Dan Walsh, *audit2allow(1)*, Linux user's manual. [Online]. Available: http://linux.die.net/man/1/audit2allow.

[49] Snapcraft, *Security policy and sandboxing*, 2020. [Online]. Available: https://snapcraft.io/docs/security-sandboxing (visited on 10/25/2020).

[50] Brian T. Sniffen, David R. Harris, and John D. Ramsdell, "Guided policy generation for application authors," in *SELinux Symposium*, 2006. [Online]. Available: http://gelit.ch/td/SELinux/Publications/Mitre_Tools.pdf.

[51] Eugene H. Spafford, "The Internet Worm Incident," in *ESEC '89, 2nd European Software Engineering Conference, University of Warwick, Coventry, UK, September 11-15, 1989, Proceedings*, ser. Lecture Notes in Computer Science, vol. 387, Springer, 1989, pp. 446–468. DOI: 10.1007/3-540-51635-2_54.

[52] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, Dave G. Andersen, and Jay Lepreau, "The Flask Security Architecture: System Support for Diverse Security Policies," in *Proceedings of the 8th USENIX Security Symposium, Washington, DC, USA, August 23-26, 1999*, USENIX Association, 1999. [Online]. Available: https://www.usenix.org/conference/8th-usenix-security-symposium/flask-security-architecture-system-support-diverse-security.

[53] S. Sultan, I. Ahmad, and T. Dimitriou, "Container Security: Issues, Challenges, and the Road Ahead," *IEEE Access*, vol. 7, pp. 52 976–52 996, 2019. DOI: 10.1109/ACCESS.2019.2911732.

[54] US Department of Defense, "Trusted Computer System Evaluation Criteria," DOD Standard DOD 5200.58-STD, 1983.

[55] David A. Wagner, "Janus: An Approach for Confinement of Untrusted Applications," M.S. thesis, University of California, Berkeley, 1999. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/1999/CSD-99-1056.pdf.

[56] Robert N. M. Watson and Jonathan Anderson, *capsicum(4)*, FreeBSD user's manual. [Online]. Available: https://www.unix.com/man-page/freebsd/4/capsicum/.

[57] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway, "Capsicum: Practical Capabilities for UNIX," in *Proceedings of the 19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010*, USENIX Association, 2010, pp. 29–46. [Online]. Available: https://www.usenix.org/legacy/event/sec10/tech/full_papers/Watson.pdf.

[58] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Kroah-Hartman, given=Greg, giveni=G., "Linux Security Modules: General Security Support for the Linux Kernel," in *Proceedings of the 11th USENIX Security Symposium, San Francisco, CA, USA, August 5-9, 2002*, USENIX, 2002, pp. 17–31. [Online]. Available: http://www.usenix.org/publications/library/proceedings/sec02/wright.html.