ISSN 2305-7254 PROCEEDING OF THE 35TH CONFERENCE OF FRUCT ASSOCIATION

Survey of Real-World Process Sandboxing

Arto Niemi
Huawei Technologies Oy (Finland) Co Ltd.
Helsinki, Finland
arto.niemi@huawei.com

***Abstract*—Attackers often exploit vulnerabilities in network-facing processes to gain access to the rest of the system. To combat this, modern operating systems such as Android, iOS and major Linux distributions allow running vulnerable or untrusted processes inside sandboxes – confined execution environments, where access to resources is restricted according to an implicit or configurable security policy. Major building blocks of sandbox implementations include namespace virtualization, system call interposition and kernel subsystem hooking. In this paper, we survey the state-of-the-art in process sandboxing, focusing on solutions that are widely deployed in consumer devices and cloud servers.**

I. Introduction

Sandboxing – also known as *confinement*, *containment* or *jailing* – restricts the resources available to a process, protecting sensitive data and reducing the attack surface of the operating system (OS) and system services. Today, all major commercial OSes use sandboxing, especially to contain network-facing applications, such as web browsers, media codecs and messagers [1]. The practical importance of sandboxing is exemplified by the high rewards offered for zero-day exploits that escape or bypass the sandbox [2]. A properly implemented and configured sandbox can prevent an attacker, who manages to compromise a network-facing application, from harming the rest of the system [3].

In this paper, we survey how process sandboxing is implemented and used in commercial devices, such as smartphones, PCs and cloud servers. In contrast to existing surveys that cover sandboxing on a single OS [4], or lack significant detail on individual sandboxes [5], [3], we aim for sufficiently deep coverage of implementation and usability aspects to provide practical aid to developers of system services and applications looking to implement the principle of least privilege.

While historically operating systems have focused on restricting actions of the human user – with the understanding that a process is always a fully authorized agent of the user – the common view today is that process containment is at least as important [6, p. 19] and that processes should be regarded as security principals of their own [7, p. 9]. Correspondingly, our focus is on *process sandboxing*, i.e. containment of running programs that are scheduled and managed by the OS directly. We do not discuss in detail user sandboxing or system-wide containment methods such as virtual machine machine based isolation like the Windows Defender Application Guard [8, p. 10]. Neither do we cover sandboxes in the form of interpreted and just-in-time compiled code, such as the Java Virtual Machine (JVM) or the Android runtime (ART).

Within process sandboxing, we distinguish three main approaches: i) process self-containment, ii) process-wrapping and iii) mandatory process containment. In these, the sandbox is configured by the process itself, by a sandbox manager application and by a privileged system administrator, respectively. In the surveyed sandbox schemes, we note important differences in how the restrictions are implemented, where the access control decisions made, how policy is defined and whether configuring the security policy requires administrator privileges. An interesting development is the emergence of unifying frameworks, such as *minijail*, that abstract multiple process sandboxing technologies and provide a convenient interface – potentially making sandboxing more approachable to non-security experts.

The rest of the paper is organized as follows. After carefully defining sandboxing and highlighting its differences to other isolation methods, we proceed by describing the basic building blocks of a sandbox, such as virtualization, system call interposition and access control, and then show how these are combined in practice to construct process sandboxes. We survey sandbox implementations on Linux, Android, macOS and iOS and Windows, covering the main operating systems used on consumer devices and servers today. We round off the survey by describing the minijail library and command-line tool, which unifies several sandbox schemes. For process self-containment sandboxes, we provide example code, and for other sandboxes we provide example policies. We conclude by discussing the pros and cons of the surveyed sandboxes, based on appraisals in the literature and our own experience of using them. Finally, we discuss interesting research directions and potential for further work.

II. What is a sandbox?

The Encyclopedia of Cryptography and Security defines sandboxing as *a technique for enforcing security policies on untrusted guest applications in a secure environment to eliminate risk to host system* [9, pp. 1075–1078]. We dissect this definition into three key components:

1) A sandbox **enforces security policies**: a sandbox aims to guarantee well-defined security properties. The goal is security, not just isolation or ease-of-use.
2) Sandboxed applications are **untrusted**: they are assumed to be malicious by design, faulty, or to contain vulnerabilities that allow untrusted entities to influence their behavior.

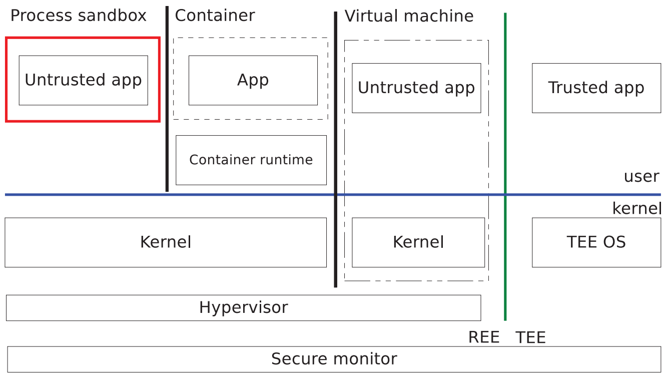

Fig. 1. Illustration of common isolation technologies. Red outline = process sandbox. Black lines = virtual memory based isolation. Blue line = user/kernel mode isolation. Green line = REE/TEE isolation.

3) A sandbox protects **the host system** from the sandboxed application. The other direction – protecting the application from host – is not a goal.

Sandboxing can be regarded as a special case of *security isolation* [5], of which Bursell distinguishes three types: 1) workload-from-workload, 2) host-from-workload and 3) workload-from-host isolation [10, pp. 202-209]. Bursell's *workload* has larger scope, but also covers our *application*. While not explicitly stated in our above definition of sandboxing, we assume "protecting the host" to also imply protecting other applications. Thus, sandboxing provides isolation of type 1 and 2. Next, we highlight differences between sandboxing and some common isolation technologies, illustrated in Fig 1.

Trusted execution environments (TEEs) provide hardware-based type-3 isolation of trusted applications (TAs) from an untrusted host platform, which in this context is called the rich execution environment (REE). TEEs are usually implemented as protected enclaves within main CPU. In TEE-based isolation, both the direction of the protection and the trust assumption are inverted compared to sandboxing. [11]

Virtual machines (VMs) can be defined as complete compute environments with their own isolated processing capabilities, memory and communication channels [12, p. 4]. The comprehensive isolation provided by a VM effectively constitutes a (large) sandbox, but with a different protection boundary: a VM does not, on its own, protect the virtualized system from applications running inside. Thus, sandboxes may still be required within the VM.

Containers, such the `docker` toolkit, focus on easing application deployment by virtualizing key platform resources, while sharing the same kernel [12, p. 4]. An application may be programmed to use a specific resources, such as particular TCP port, which may not always be free on the host. A container can then map the application's preferred resource name (such as a port number) to another resource that is actually available. Containers do not generally attempt to contain a malicious application; instead, containers focus on preventing resource conflicts and ensuring the correct versions of dependent libraries are available to the application [13].

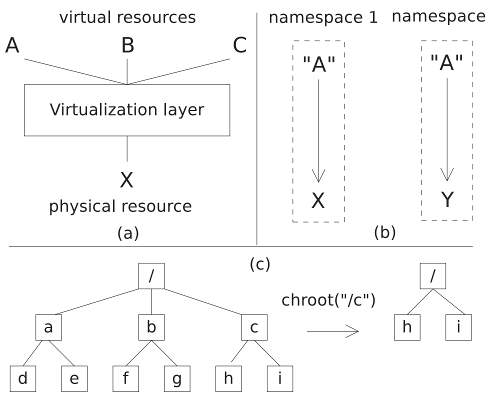

Fig. 2. Common sandbox building blocks. a) Virtualization via multiplexing allows a single physical resource to be accessed via multiple virtual resources. b) Namespaces allow using the same name for multiple resources. c) The chroot system call changes the meaning of the root directory for a particular process, restricting the process' view of the filesystem.

III. Building blocks of a sandbox

A. Virtualization

Virtualization forces a resource to be accessed via an abstraction layer. The layer exposes a *virtual* resource that is – from the user's perspective – identical to the underlying *physical* resource. Virtualization can be implemented with three basic techniques: multiplexing, where a single physical resource is accessed via multiple virtual resources (Fig. 2a), aggregation, where multiple physical resources are accessed via a single virtual resource) and emulation. [12, pp. 1-3]

Virtual memory isolates processes by preventing them from accessing each other's memory. Other resources can also be virtualized with sandboxing in mind; a well-known example of this is the Unix *namespace* concept, which virtualizes, for example, filesystems and network interfaces [14]. Namespaces were pioneered by the Plan 9 operating system [14], which the took the Unix concept of "everything is a file" to the extreme. Resources such as devices, network interfaces, etc. were represented as files in single-rooted file system hierarchies. There was no "global" filesystem, but processes could assemble private views of the system by constructing a "name space" file that connects the resources.

The mount namespace was added into Linux in 2002 and followed later by UTS (hostname), process (PID), network and cgroup namespaces. Namespaces, along with chroot and cgroups, are the main technologies underlying popular container frameworks such as docker [13]. A process is always in exactly one namespace of every kind, and can only see and use resources in its own namespace. [15] Having multiple instances of a namespace allows having two resources with the same name [16], as illustrated in Fig. 2b. By default, a process inherits the namespaces of its parent, but a process may choose to have its own unique namespaces. On Linux this can be done with `unshare` system call.

B. Access control

Sandboxes commonly make use of available access control frameworks, including the original Unix scheme based on user (instead of process) identity [17]. Access control methods can be classified as *discretionary* or *mandatory*, depending on whether the subjects (users, processes) are allowed to influence policy.

A persistent issue with access control on Unix-like systems is the prevalent usage of the all-powerful "root" account privileges. A process with root privileges can override access control decisions, execute any binary and even load kernel modules – thus, a root-privileged process can execute arbitrary code both in user and kernel space. Most network-facing daemons are still running as root; a single vulnerability in these can compromise the entire system [18].

Capabilities, defined in the POSIX.1e standard, are widely used in Linux to break the omnipotent root user's privileges into more fine-grained components. There are currently over 30 capabilities in the Linux kernel [13, p. 19]. Examples include `CAP_SYS_BOOT`, which allows rebooting the system, `CAP_DAC_OVERRIDE`, which allows overriding DAC access control checks, and the (unfortunately) quite wide-ranging `CAP_SYS_ADMIN` which allows a multitude of system administrator actions, such as looking up file extended attributes or loading Berkeley Packet Filter (BPF) programs (see Section VII-A2) into the kernel with the `bpf` system call.

C. System call interposition

Except for some low-level embedded systems, a process interacts with its environment almost exclusively via the OS. Thus, the system call interface is a logical place to enforce the sandbox' security policy. In particular, system calls are the only way a process can access resources outside of its own virtual memory range. *System call interposition* refers to a method where system calls are intercepted close to their entry point. Execution is transferred, via hooks, to a policy enforcement point (PEP) that decides whether to allow the call. The PEP may be implemented either in kernel or userspace. Despite its seeming simplicity, complexity of modern system call interfaces make system call interposition is difficult to implement securely. Examples of common issues include overlooking indirect paths to resources, symbolic link or relative path races – and even unexpected side-effects of denying system calls [19]. System call interposition also suffers from relatively low granularity [20] and may require duplication of operating system functionality – for example, pathname parsing may need to be reimplemented if pathname resolution at the system call entry point is desired.

D. Kernel subsystem hooking

The full context and details of a system call are not always known at its the entry point. For example, symbolic links are resolved later. Reading from a resource given its file descriptor (an opaque integer) is another example: whether the descriptor refers to a standard storage file, a device or a socket, is

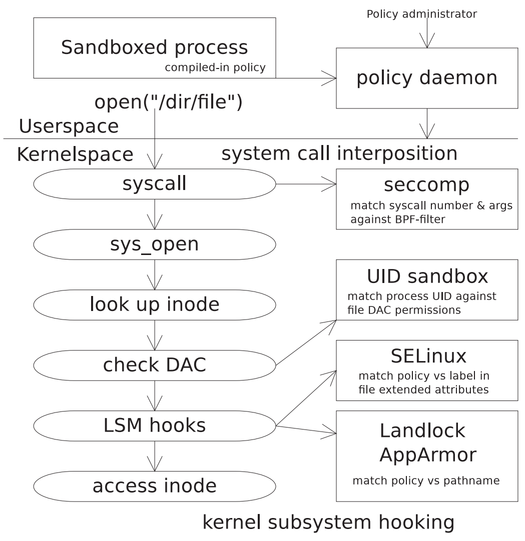

Fig. 3. Illustration of common sandbox implementation techniques on Linux: system call interposition, discretionary access control and kernel subsystem hooking. Each sandboxing mechanism checks whether the system call or access to kernel object is allowed by policy. If not, an error code is returned or the process is killed. The policy is either stored in the application binary or configured by a privileged policy administrator. Adapted from [20].

not known at the start of the `read` system call. A popular approach is therefore to insert hooks closer to the specific action that the system call triggers. Well-known examples of this approach include Linux security modules (LSM) [20] and the TrustedBSD mandatory access control framework (MACF) [21].

E. System call wrapping

In *system call wrapping*, the process is not allowed access the OS directly, but must, instead, use a "wrapper" API provided by the sandbox. For example, Java programs can only call access the OS via the Java Virtual Machine. Sandbox policy enforcement can then be done in the wrapper. This approach is sometimes called process-level virtualization.

F. Process self-containment

A process can contain itself by voluntarily giving up privileges it does not need. In practice, the process does this by invoking special self-containment system calls. For example, a process may start with root privileges, and invoke the `capset` system call to reduce privileges to the bare minimum the process needs. It is implicitly assumed that the process is trusted until the point where the privileges are dropped and untrusted afterwards. The `seccomp` system call allows the process to specify a filter that restricts its access to further system calls. Landlock [22], a recent Linux Security Module (LSM), provides three self-containment system calls (`landlock_create_ruleset`, `landlock_add_rule` and `landlock_restrict_self`). In contrast to most earlier self-containment technologies, Landlock does not require the calling process to be privileged in order to use the API.

G. Process-wrapping

In process-wrapping, the sandboxed process is started and its privileges are restricted by another process – a sandbox manager. Examples of process wrappers include command-line tools such as `jailkit`, `firejail`, `minijail0`, AppArmor's `aa-exec` the SELinux-based `sandbox` and macOS' deprecated `sandbox-exec`. In some setups, the `init` daemon functions as a sandbox manager. We note that process-wrapping and process self-containment are not always clearly delineated. For example, a process can contain itself by dropping privileges and then invoking the `exec` system call to start another process. This corresponds to process-wrapping if the new process inherits the restrictions that were configured by the parent.

H. Policy management

Policy management has often been the Achilles' heel of "post-DAC" sandboxes. While the UID-based DAC sandbox is simple enough not to require a separate policy definition, this is not the case for most more complex sandboxes – these suffer from what Anderson calls *dual coding of policy* [4, p. 3]: first, the code describes that a program does, and then a separate policy describes what the program is allowed to do. Sandbox mechanisms that require a separate policy, such as SELinux and AppArmor, are notorious for being difficult to use in practice [23].

Many sandboxing schemes provide a domain-specific policy language, often also allowing the policy to be compiled into binary when performance is critical. Another approach is to allow policy configuration at run-time via an API such as system calls. Whether sandboxed processes are able to influence the policy constitutes the classic distinction between discretionary (DAC) and mandatory access control (MAC). In DAC-based sandboxing, a process may be allowed, for example, to configure the policy for its child processes or for created files. In MAC-based sandboxing, subjects have no say over policy configuration.

Some sandbox technologies require access-controlled resources to be *labeled*, i.e. assigned an additional name by which they can be referred to in the security policy. A policy that operates on existing names, such as absolute pathnames (such as `/etc/passwd`) does not require labeling.

IV. History of sandboxing

Protection was a major design goal in early operating systems such as Multics. Important isolation primitives, such as virtual memory and capability-based addressing were invented during this period, ranging roughly from the 1960s to the mid-1970s. According to Anderson [4, p. 1], however, this laudable focus on security was lost in the transition to Unix dominance. Unix [17] initially had few of these security features. Its security model (DAC) accounted for malicious users, but not for malicious (or vulnerable) processes started by a trusted user. Since important processes such as network daemons and remote login services ran with privileges of the all-powerful root user, compromising one these was enough to compromise the entire system.

Addition of the `chroot` system call into Unix Version 7 in 1982 [24, p. 11] can be regarded as the first step towards modern process sandboxing. Invoking `chroot` permanently changes the process' root directory. This, in theory, restricts the process to files below the new root. However, sandboxing was not an original goal of `chroot`, and many practical bypass techniques were found over time. The main problem is that `chroot` only controls file access based on the pathnames. So, for example, access by file descriptor is not controlled. [25]

FreeBSD's `jail` utility was one of the earliest process-wrapper sandboxes. In addition to improved chroot-like filesystem restrictions, `jail` also restricted the jailed process' visibility to other processes and network interfaces [24]. Janus [26], [27], developed mostly in the late 1990s, was one of the first system call interposition based sandboxes in the 1990s. In an important later work [19], the authors reflected upon difficulties they experienced trying to implement system call interposition without loopholes. The work of Provos et al [28] in 2003 was an early effort to provide a self-containment sandbox on Unix like systems. Following the principle of *privilege separation*, the authors proposed to split the SSH daemon, which ran as root, into two processes: a monitor and a slave. Only the monitor would retain root privileges. To serve new connections, the monitor would fork off a slave process under a unique UID. The slave would then invoke chroot to restrict its access to the file system, and drop privileges. When a privileged action was required, the slave would request this as a service from the monitor via a well-defined interface.

Today, as we shall see in the rest of this paper, sandboxes are widely deployed in commercial devices such as smartphones and PCs.

V. Framework

For each of the studied real-world process sandboxes, we attempt to answer the following questions:

- **Implementation**. Who enables the sandbox, a sandbox manager or the sandboxed process itself? Which technologies is the sandbox implemented with? At what point in the kernel are the access permission checks done?
- **Policy**. Does the sandboxing scheme provide its own policy language? Does the sandbox require access-controlled object to be labeled separately?
- **Privileges**. Does configuring the policy and enabling the sandbox require administrator privileges?
- **Coverage**. What operations can be restricted by the sandbox? Does it allow filesystem-based and network interface restrictions?

VI. Process sandboxing in mobile devices

A. Apple

The Apple sandbox, originally called *Seatbelt*, first appeared in the OS X 10.5 in 2007 [29, Chapter 5] and became significantly easier use to when the developer-friendly *App Sandbox* interface was introduced in OS X 10.7 in 2011 [30, p. 151]. Today, sandboxing is used on all Apple devices [31], [32].

Over time, Apple has incrementally tightened the sandbox, partly in response to high-profile vulnerabilities, such as the jailbreakme.com attack, where users could gain root privileges on an iPhone by simply visiting a website with the Safari browser [33, p. 3]. For example, between 2019 and 2022, Apple expanded the sandbox with e.g. ability to filter Unix and Mach system calls [32, pp. 11–12].

The inner workings of Apple's sandbox are closed-source and opaque, but reverse engineering efforts have shed some light on it [34]. It is known to be based [35, p. 1] on the TrustedBSD mandatory access control framework (MACF) [21]. The MACF is implemented in the XNU kernel, providing over 200 policy hooks [36] and kernel object tagging [37, p. 5], with the ability to restrict access filesystem, system call, interprocess communication (IPC) and network interfaces, among others. The sandbox policy is enforced in the `Sandbox.kext` kernel extension [37, p. 14]. A userspace library, `libsandbox`, provides a policy compiler and process self-containment APIs. The `sandboxd` daemon is used for logging and tracing.

There are two ways to configure the sandbox policy on Apple devices: Sandbox Profile Language (SBPL) and entitlements. iOS allows both, but macOS only supports entitlements. During application loading, the policy is extracted, compiled into binary and then stored in the kernel.

SBPL is a Scheme-like language. It was originally the only way to configure the sandbox before the App Sandbox framework. It is still supported and used extensively by Apple's own applications and system daemons. The following example, paraphrased from [35], allows reading files in the /usr directory, except for one particular file. It also allows connecting to the TLS port of a particular server. The first line makes this a whitelist policy; all other actions are denied.

```
(deny default)
(deny file-read* (literal "/usr/forbidden_file"))
(allow file-read* (regex #"/usr/*"))
(allow network-outbound (remote tcp "server.com
    :443"))
```

In earlier iOS versions, the compiled SBPL policies were stored in a separate location (`/usr/libexec/sandboxd`). Since iOS 9, all compiled policies are stored in single binary blob in the sandbox kernel extension (`com.apple.security.sandbox`) [35, p. 6] in read-only memory [31, p. 2]. The format of the blob is closed-source, but it has been reverse-engineered by researchers (e.g. [35]). The default sandbox policy, called *container*, has also been reverse-engineered back to the human-readable SBPL format [38].

Entitlements are a more developer-friendly way to configure the sandbox. They are key-value pairs the developer can write either in XML or via a GUI in the Xcode IDE. Entitlements are stored in the application's executable file, and covered by its signature, protecting them against tampering. The entitlement keys all start with the prefix `com.apple.security`. For example, `com.apple.security.network.server.client` allows the application to connect to a server over TCP/IP, and `com.apple.security.device.camera` allows the app to use the camera. [39]

In iOS, the default *container* system-wide sandbox policy is applied to all applications, but it is possible to define a complementary per-process policy [31, p. 13]. On macOS, sandboxing is mandatory for applications distributed via the official app store, but optional for others. To enable the sandbox for a macOS application, the developer can either include the `com.apple.security.app-sandbox` entitlement in the binary, or invoke the sandbox by calling the `sandbox_init` API offered by `libsandbox`, with the SBPL policy as a parameter. Thus, the Apple Sandbox provides mandatory process containment for iOS apps, but for macOS apps, also a self-containment option is provided. In addition, Apple provides the (now deprecated) `sandbox-exec` process wrapper. [30]

B. Android

From the start, Android has sandboxed apps and system services using an approach based on traditional Unix DAC [4, pp. 2–3]. Each app is assigned a separate UID and a directory owned by the app [40]. A limited set of UID sandboxes are used for system services. For example, media frameworks run under a single UID called the `AID_MEDIA`. UIDs are still the primary application sandboxing mechanism in Android, but they have recently been complemented with seccomp and SELinux. Since Android 8, all applications run with a seccomp filter, and Android 9 added a per-app SELinux sandbox [7, pp. 17-31]. Android 12 added *Private Compute Core* (PCC) for sandboxing the processing of private data, such as sensor readings, within an application [7, p. 25]. SDK sandboxing became possible in Android 13 with the *ads SDK runtime*, which allows apps to load code from third-party libraries such that loaded code runs in a sandbox that is separate from the app sandbox [7, p. 22]. This allows SDKs to be distributed and updated separately from apps, and protects against SDK abuse (e.g. programmatic clicks in advertising SDKs). Users can also give an app permission to access their location, without giving the SDK the same permission.

C. Windows

In Windows, processes are restricted using security tokens. These contain security identifiers (SIDs) that uniquely identify the process and its owner. Windows 8 added the *AppContainer* sandbox, originally codenamed *LowBox*. All Universal Windows Platform (UWP) apps are sandboxed with AppContainer, and regular desktop applications can also be configured to use it [41]. In particular, the Edge browser relies heavily on AppContainer sandboxing [8]. The internals of AppContainer are closed-source and largely undocumented, although some information can be found in [42, p. 684–709] and Microsoft's online documentation, e.g. [43].

The sandboxed process receives a SID that combines the identity of the user and the application. This means, for example, that the process no longer gets the same level of access as the user who started it. An AppContainer-sandboxed process receives its own private, restricted view (i.e. a namespace) of the filesystem, the registry and kernel objects. For example, based on the string representation of its SID (x) a sandboxed process receives private directory Sessions\x\AppContainerNamedObjects, listing the kernel objects it can access. [42]

AppContainer adds the notion of capability into security tokens, allowing more fine-grained sandbox policies. For example, the internetClient capability allows outgoing Internet connections. The application's capabilities are declared in its package manifest. Enabling any capability in a sensitive subset called *restricted capabilities*, requires approval from Microsoft before the app can be accepted into the App Store. These include, for example, inputObservation, which the app to observe various forms of raw input, such as keyboard events and mouse movements and locationHistory, which allows the app to access location history of the devices. [44]

VII. Process sandboxing on Linux PCs and servers

A. Seccomp

Secure computing mode (seccomp) is a process self-containment feature in the Linux kernel, implemented with system call interposition. Seccomp policies are configured and enabled with the seccomp system call. [45]

1) seccomp-1: The first version of seccomp, often called *strict seccomp* or *seccomp-1*, was added into the mainline Linux kernel in 2005. When enabled at runtime, seccomp-1 restricts the application to only four system calls: read, write, exit and sigreturn. Since open is not among these, the application needs to open the files it wishes to access *before* enabling seccomp-1 mode. Unfortunately, modern versions of the gcc compiler use exit_group instead of exit. Such programs will be killed at the end when seccomp denies the exit_group call.

2) seccomp-bpf: The successor of seccomp-1, called *seccomp-bpf* or *seccomp filter mode*, was merged into the mainline kernel in 2012. It is much more versatile than its predecessor: instead of a simple "deny-list", seccomp-bpf makes access control decisions programmable, using the classic Berkeley Packet Filter (cBPF) assembly-like language. The cBPF code (a filter) is interpreted by the kernel on system call entry. The result is either that the system call is allowed, the process is killed, or a (possibly filter-defined) error code is returned. seccomp-bpf makes it possible to deny a system call *conditionally*, based on its arguments. For example, one can prevent an application from writing to any other file except standard output by requiring the fd argument to be 1. However, pointers cannot be dereferenced in a cBPF filter, so inspecting non-integer syscall arguments is difficult. For example, the first argument to open is a C string – a pointer to char – so it is not possible to use a cBPF filter to allow or forbid the process to open a particular file.

3) seccomp-ebpf: Jia et al. [46] discuss several further drawbacks of seccomp-bpf. Since cBPF filters are stateless, it is not possible to use them to apply count-limiting to system calls. For example, it is not possible to allow a container to call exec only once or to limit the number of files a process may open using the open system call. It is not possible to invoke any Linux kernel functionality in the cBPF filter, which prevents e.g. accessing a timer to check when the system call was last invoked, so rate-limiting with cBPF is also not possible. It also not possible to invoke other cBPF filters. To address such issues, Jia et al. propose *seccomp-ebpf*, an alternative implementation of seccomp that uses the extended BPF language (eBPF) instead of cBPF. Although seccomp-2 also invokes eBPF under the hood, it is not possible to use eBPF to specify filters.

4) seccomp notifier: Seccomp notifier is an earlier attempt to solve similar issues as seccomp-ebpf. Seccomp notifier introduces a userspace agent to complement cBPF filters. The userspace agent can be stateful, but it introduces some amount of overhead due to the frequent user-kernel space context switches during seccomp evaluation. seccomp-notify was added in Linux 5.0 in 2019.

5) The seccomp API: A seccomp filter is loaded using the seccomp system call, whose C interface is:

```
int seccomp(
  unsigned int operation,/* e.g.
      SECCOMP_SET_MODE_FILTER */
  unsigned int flags,
  void *args // The BPF program to load

);
```

The filter is defined as a BPF program, consisting of a sequence of statements with the following format:

```
struct sock_filter {
 __u16 code; // Instruction code,
             // e.g. "BPF_JMP | BPF_JEQ"
 __u8 jt; // Distance of forward jump on 'true'
 __u8 jf; // Distance of forward jump on 'false'
 __u32 k; // Generic multiuse field
};
```

Each statement operates on a variable of the following type, which represents the system call invocation to be filtered:

```
struct seccomp_data {
  int nr; // System call number
  __u32 arch;
  __u64 instruction_pointer;
  __u64 args[6]; // System call arguments
};
```

The following code installs a seccomp filter that allows the system calls which a minimal C program needs to open and write to a file. For simplicity, we assume the system call number is always correct for the current architecture.

```
struct sock_filter filter[] = {
 // Load syscall number from seccomp_data
 // into accumulator
 BPF_STMT(BPF_LD|BPF_W|BPF_ABS,
    (offsetof(struct seccomp_data, nr))),
 // Jump to allow if syscall number is
 // 1 (write). 2 (open), 5 (fstat)
 // 12 (brk) or 257 (openat)
 BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 1, 5, 0),
 BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 2, 4, 0),
 BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 5, 3, 0),
 BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 12, 2, 0),
 BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 257, 1, 0),
 // Return and kill the process
 BPF_STMT(BPF_RET|BPF_K,SECCOMP_RET_KILL),
 // Return and allow the system call
 BPF_STMT(BPF_RET|BPF_K,SECCOMP_RET_ALLOW)
};
struct sock_fprog prog = {
  .len = (unsigned short) \
     (sizeof(filter)/sizeof(filter[0])),
  .filter = filter
};
// Allow seccomp without CAP_SYS_ADMIN
prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0);
// Register the filter
rc = syscall(SYS_seccomp,
             SECCOMP_SET_MODE_FILTER,
             0, &prog);
```

libseccomp simplifies the specification of seccomp filters. For example, the following creates a simple whitelist filter that corresponds to seccomp-1, except that exit_group is allowed instead of exit:

```
# include <seccomp.h>
int main(int argc, char **argv) {
  scmp_filter_ctx ctx = seccomp_init(
             SCMP_ACT_KILL_PROCESS);
  int rc = 0;
  rc |= seccomp_rule_add(ctx,
         SCMP_ACT_ALLOW,
         SCMP_SYS(exit_group), 0);
  rc |= seccomp_rule_add(ctx,
         SCMP_ACT_ALLOW,
         SCMP_SYS_(read), 0);
  rc |= seccomp_rule_add(ctx,
         SCMP_ACT_ALLOW,
         SCMP_SYS_(write), 0);
  rc |= seccomp_rule_add(ctx,
         SCMP_ACT_ALLOW,
         SCMP_SYS_(sigreturn), 0);
  if (rc == 0) {
    rc = seccomp_load(ctx);
  }
  if (rc == 0) {
    // Rest of main()
  }
}
```

B. SELinux

SELinux, developed by the National Security Agency (NSA) together with other security research organizations, is a MAC framework included in several widely-used Linux distributions. It was released as open source in 2000 and merged into the mainline Linux kernel in 2003. It is the default MAC implementation on Red Hat Linux [47, Chapter 24], Fedora and Android.

SELinux supports role-based access control (RBAC), type enforcement (TE) and multi-level security (MLS), although MLS is rarely used in consumer devices. TE is based on labeling objects with *types*, assigning subjects to *domains* and *roles*, and providing rules (policies) that allow certain domains and roles to access certain types [47, p. 671]. SELinux policies are written in terms of subjects, objects and actions. Subjects, typically running processes, carry out actions on objects, which a usually files or other processes.

SELinux requires each object to be labeled with a security context, a colon-delimited string of the form username:role:type:mls-range. Of these, *type* is most important in practice. Android, for example, sets mls-range to a constant value s0, and role to either r (for processes) or object_r (for objects). Security context can be assigned during filesystem initialization or implicitly when objects are created: files and processes inherit the security context of their parent directory or process, respectively. The policy can also specify domain transition rules to configure the automatic assignment of security contexts. On Linux-based systems, security contexts are stored in the extended attributes of each file. Extended attributes are essentially arbitrary, but length-limited data that is not interpreted by the filesystem. The ext4 filesystem allows extended attributes in the form of key-value pairs; SELinux security context is stored under the key *security.selinux*. The filesystem itself ignores the attributes, but the kernel uses them for access control decisions. The security context of a file can be retrieved with the lgetxattr system call (use e.g. by the ls -Z command) and process' security contexts via the a special file in the proc filesystem (/proc/self/attr/current) [48, Chapter 12]

As one of the oldest and most popular MAC frameworks, SELinux benefits from a rich set of supporting tools. These include, for example, the sandbox process wrapper and audit2allow, which automatically converts logged SELinux denials into rules that would prevent the denials. A major challenge with using SELinux is its difficulty. Labeling every relevant resource, creating domains and roles, and especially learning the policy language syntax and working with the policy tools is widely regarded to have a deep learning curve [47, p. 688].

C. TOMOYO

TOMOYO, short for *Task Oriented Management Obviates Your Onus on Linux*, was added into the mainline Linux kernel in 2009. The first version (1.0) was an independent MAC implementation, but version 2.0 was reworked into an LSM. In contrast to SELinux, TOMOYO does not require all protection targets to be labeled. Instead of labels, the canonical pathnames of files are used in policies. Thus, the administrator need not separately label each file. In TOMOYO, every process in the system belongs to a domain. The difference to SELinux is that the process' domain is determined based on its execution history instead of manual assignment (labeling). Key benefits of TOMOYO include a simple policy language and automatic policy generation based on the observation of a running process. [49]

The following policy grants a domain read access to all files in /tmp/myprog, read-write access to a particular file, and allows outgoing connections to the TLS port of the loopback IP address:

```
file read /tmp/myprog/*.txt
file write /tmp/myprog/rwfile.txt
network inet stream connect 127.0.0.1 443
```

D. Landlock

Landlock [22] is a Linux security module (LSM) for process self-containment. A major benefit is that Landlock does not require any extra capabilities, so even a minimally-privileged process can use the Landlock syscalls to contain itself. Landlock currently supports mostly filesystem-based restrictions. A recent patch added support for restricting access to TCP ports.

To restrict itself using Landlock, a process must first create a ruleset using the `landlock_create_ruleset` system call. The scope of the ruleset, i.e. which objects are to be subjected to access control, is provided as an argument; Landlock's default action for all of these is deny. The call returns a handle to the created ruleset. Next, the process calls `landlock_add_rule` to add allow rules. There are two types of rules: filesystem rules and network rules. Filesystem rules apply to a subtree in the filesystem hierarchy, whose root is specified with file descriptor. Network rules apply to sockets and TCP/UPD port numbers. Finally, `landlock_restrict_self` enables the enforcement of the ruleset on the process and its future children permanently.

Landlock currently does not support restricting non-filesystem based system calls. For example, `execve` calls cannot be prevented with Landlock. Support for network-based restrictions is also limited. Currently it is only possible to prevent connecting or binding a socket to a particular port. It is also important to note that Landlock rules are applied to file descriptors, so a process must open the file or directory first before adding allow rules.

The following minimal example allows a process to read files from `/usr/myprog`. Opening files for writing is denied. To keep the example code short, all error checking is omitted.

```
__u64 all_fs_rules =
    LANDLOCK_ACCESS_FS_EXECUTE |
    LANDLOCK_ACCESS_FS_WRITE_FILE |
    LANDLOCK_ACCESS_FS_READ_FILE |
    LANDLOCK_ACCESS_FS_READ_DIR |
    LANDLOCK_ACCESS_FS_REMOVE_DIR |
    LANDLOCK_ACCESS_FS_REMOVE_FILE |
    LANDLOCK_ACCESS_FS_MAKE_CHAR |
    LANDLOCK_ACCESS_FS_MAKE_DIR |
    LANDLOCK_ACCESS_FS_MAKE_REG |
    LANDLOCK_ACCESS_FS_MAKE_SOCK |
    LANDLOCK_ACCESS_FS_MAKE_FIFO |
    LANDLOCK_ACCESS_FS_MAKE_BLOCK |
    LANDLOCK_ACCESS_FS_MAKE_SYM |
    LANDLOCK_ACCESS_FS_REFER |
    LANDLOCK_ACCESS_FS_TRUNCATE;
struct landlock_ruleset_attr rules =
{
 .handled_access_fs = all_fs_rules,
 .handled_access_net = 0,
};
__u64 allowed_fs_rules =
    // Allow reading files and directories:
    LANDLOCK_ACCESS_FS_READ_FILE |
    LANDLOCK_ACCESS_FS_READ_DIR |
    // Allow creating regular files:
    LANDLOCK_ACCESS_FS_MAKE_REG;
struct landlock_path_beneath_attr path_beneath =
{
 .allowed_access = allowed_fs_rules,
 .parent_fd = 0,
};
int rc, fd, ruleset_fd;

// Specify sandbox scope
ruleset_fd = landlock_create_ruleset(
  &rules, sizeof(rules), 0);
```

```
// Add allow rules. These will applied to the
// filesystem subtree rooted at "/usr/myprog"
// Access to other parts of the filesystem
// will be denied.
fd = open("/usr/myprog", O_PATH | O_CLOEXEC);
path_beneath.parent_fd = fd;
rc = landlock_add_rule(
  ruleset_fd,
  LANDLOCK_RULE_PATH_BENEATH,
  &path_beneath, 0);

// As with seccomp, this is needed:
rc = prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0);

// Enable sandbox enforcement
rc = landlock_restrict_self(ruleset_fd, 0);

// Should succeed:
fd = open("readfile.txt", O_RDONLY | O_CREAT);
assert(fd != -1);

// Should fail:
fd = open("writefile.txt", O_WRONLY | O_CREAT);
assert(fd == -1);
```

E. AppArmor

AppArmor is an LSM-based mandatory access control framework, developed by Novell and introduced into Linux in 2007. Since 2009, the project has been supported by Canonical. AppArmor is currently the default MAC framework in Ubuntu distributions. [50]

AppArmor relies centralized policy configuration and storage, like SELinux. Instead of labels, AppArmor, like Landlock, uses pathnames, so the laborious labeling step is not needed. AppArmor is based on type enforcement and does not provide role-based access control or multi-level security. AppArmor's policy language is simpler than that of SELinux. Subjects (processes) are identified by the pathname of their binaries. The pathname is made unique by resolving links and constructing a canonical pathname against the root mount namespace. [51]

AppArmor profiles are written in a custom language and stored by default in `/etc/apparmor.d`. Before a policy can be used, it must be compiled into binary and loaded into the kernel using the `apparmor_parser` tool. AppArmor provides a rich set of command-line tools. For example `aa-genprof` can, like SELinux's `audit2allow`, automatically a profile for a given program based on the denials logged by AppArmor in system logs. For process-wrapping, AppArmor provides the `aa-exec` utility [52].

The following example policy grants the process whose binary pathname is `/usr/bin/myprog` the read and write rights to all files in `/tmp/myprog`, except for `readonlyfile.txt`, for which only read permission is granted:

```
/usr/bin/myprog {
  /usr/bin/myprog mr,

  owner /tmp/myprog/*.txt rw,
  deny /tmp/myprog/readonlyfile.txt w,
}
```

The profile can be enforced by saving it to `/etc/apparmor.d/usr.bin.myprog` and invoking `sudo aa-enforce /usr/bin/myprog`.

VIII. Minijail

The minijail open source project [53] provides a C library (libminijail) and a command-line tool (minijail0) for process sandboxing via self-containment and process-wrapping. Minijail provides a unified abstraction layer on top of several Linux kernel features, including namespaces, cgroups, rlimit, chroot, POSIX capabilities, seccomp and Landlock.

For process-wrapping, the minijail project provides the minijail0 tool [54], which supports a rich set of command-line options for configuring sandbox restrictions on the started process. Since not all capabilities are inherited by child processes and because seccomp filters typically disallow execve, some restrictions must be applied *after* the jailed process has been started. This is done using a preload library (libminijailpreload), which injects code into the process that runs before the actual start function. Note that for restrictions based on chroot, capabilities, etc., the minijail0 binary needs to be started as root. Only seccomp-based restrictions can be used without extra privileges, as long as the process has the no_new_privs attribute, which can be set with prctl system call.

For process self-containment, minijail provides a single API that abstracts the underlying sandboxing primitives. For seccomp, minijail also provides a simple policy language:

```
<syscall_name>:<ftrace filter policy>
<syscall_number>:<ftrace filter policy>
<empty line>
```

In the following example, we wish to sandbox a process, so that it is not able to do anything else except to open and read particular files. All other actions should be denied. The example uses the following seccomp policy:

```
open: allow
openat: allow
close: allow
read: allow
write: allow
rt_sigreturn: allow
exit: allow
exit_group: allow
nanosleep: return EACCES
```

We compile the above policy into binary (/tmp/myprog/-filter.bpf) using minijail's compile_seccomp_policy.py command-line tool. If the kernel has been compiled with CONFIG_FTRACE_SYSCALLS, the "return EACCES" policy causes nanosleep to return an error, otherwise, the process is killed when tries to invoke nanosleep.

The following example demonstrates the use of chroot, seccomp and Landlock-based filesystem allow rules. For brevity, all error code checking is omitted. The array bpf_filter contains the above seccomp policy in compiled form.

```
struct minijail *jail;
int rc;
unsigned char bpf_filter[] = {
    0x20, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00,
    0x15, 0x00, 0x01, 0x00, 0x3e, 0x00, 0x00, 0xc0,
    0x06, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x20, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x35, 0x00, 0x00, 0x04, 0x04, 0x00, 0x00, 0x00,
    0x15, 0x00, 0x03, 0x00, 0x01, 0x01, 0x00, 0x00,
    0x15, 0x00, 0x02, 0x00, 0xe7, 0x00, 0x00, 0x00,
    0x15, 0x00, 0x01, 0x00, 0x3c, 0x00, 0x00, 0x00,
    0x15, 0x00, 0x00, 0x01, 0x0f, 0x00, 0x00, 0x00,
    0x06, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0x7f,
    0x06, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};
struct sock_fprog filter;
int fd;
struct timespec req = { 1, 0 };
struct timespec rem = { 0, 0 };

filter.len = sizeof(bpf_filter) / sizeof(struct
    sock_filter);
filter.filter = (struct sock_filter*)bpf_filter;

jail = minijail_new();

// Restrict filesystem view
rc = minijail_enter_chroot(jail, "/tmp/myprog");

// Set Landlock-based restrictions
if (minijail_is_fs_restriction_available())
{
    minijail_add_fs_restriction_rw(jail,
        "rwfile.txt");
    minijail_add_fs_restriction_ro(jail,
        "readonlyfile.txt");
}

// Set seccomp filter
minijail_no_new_privs(jail);
minijail_set_seccomp_filters(jail, &filter);
minijail_use_seccomp_filter(jail);

// Self-contain
minijail_enter(jail);

# define try_open(file, flag)                  \
do {                                           \
    fd = open(file, flag);                     \
    printf("open(" file "," #flag "):%s\n", \
           fd < 0 ? strerror(errno) : "ok");   \
} while(0)

// Should succeed:
try_open("rwfile.txt", O_WRONLY);
try_open("readonlyfile.txt", O_RDONLY);

// Should fail:
try_open("readonlyfile.txt", O_WRONLY);
try_open("/tmp/forbidden_file", O_RDONLY);
rc = nanosleep(&req, &rem);
printf("nanosleep():%s\n",
       rc == 0 ? "ok" : strerror(errno));

minijail_destroy(jail);
```

IX. Comparison

The results of our survey are summarized in Table I. We see that mandatory process containment is a popular approach to sandboxing. It allows an administrator or the OEM to define a central security policy that restricts all processes. The main drawback of this approach is the complexity of writing and configuring the policies. Also, sandboxes based on SELinux require labeling each resource in the system that is to be access controlled, a major task on its own.

Process self-containment moves responsibility of sandbox configuration to the developer. Its major benefit is that the

TABLE I. COMPARISON OF THE SURVEYED SANDBOX SCHEMES

			Implementation			Policy	
Name	Type	Based on	Protected objects	Policy enforcement point	Storage	Labels	Privs
Android	mand	DAC, SELinux	syscall, file, ipc	DAC, LSM	cent	yes	yes
AppArmor	mand, wrap, self	LSM	syscall, file	LSM	cent	no	yes
AppContainer	mand	Windows kernel	file, net	unknown	dist	no	no
Apple Sandbox	mand, self, wrap	TrustedBSD MACF	syscall, file, net, ipc	sandbox.kext	dist	no	no
Landlock	self	LSM	file, net	LSM	dist	no	no
libminijail	self, wrap	several (note 1)	syscall, file	LSMs, syscall entry	dist	no	note 2
seccomp	self	BPF	syscall	syscall entry	dist	no	no
SELinux	mand, wrap	LSM	syscall, file, net, ipc	LSM	cent	yes	yes
TOMOYO	mand	LSM	file, net	LSM	cent	no	yes

mand : mandatory process containment
self: process self-containment
wrap: process-wrapping
syscall: system call, fs: filesystem, net: network interface, ipc: inter-process communication
dist: distributed, cent : centralized
Labels: Requires objects to be labeled, Privs: Policy configuration requires privileges
note1: libminijail uses seccomp, DAC, Landlock, chroot, capabilities and namespaces
note2: minijail requires privileges to configure chroot, DAC, capabilities and namespaces

sandbox can be configured without administrator privileges. For example, the Landlock LSM and seccomp work for unprivileged processes. This allows a distributed approach to policy configuration, e.g. including the policy in the application binary itself.

AppArmor, Apple Sandbox and libminijail also provide a process-wrapper tool, which can be used to start another process in a sandbox, with the policy configured on the command line. However, Apple's process-wrapper tool (`sandbox-exec`) is now deprecated. An SELinux-based process wrapper, simply called `sandbox`, is also available. Process wrappers are a useful way to start system daemons and educational for learning, but are less useful for developers looking to deploy their application on a remote platform.

Implementation-wise, the surveyed sandbox schemes rely heavily on MAC frameworks such as LSM on Linux and TrustedBSD MAC framework on iOS and macOS. Thus, kernel subsystem hooking seems to be the dominant approach to sandbox implementation. Android continues to use the traditional DAC to sandbox processes by providing each process a unique UID and working directory, complemented with seccomp, which is based on system call interposition.

X. CONCLUSIONS AND FUTURE WORK

The main difficulty with all sandboxing approaches is that significant expertise and time investment is needed to deploy them. While none of the surveyed sandboxing tools are particularly easy to learn, we found Landlock and minijail to be relatively convenient from a developer perspective. In general, process self-containment and process-wrapping seems to be an order of magnitude easier to configure than MAC policies. We find that although SELinux is most generic and versatile sandbox technology, it is also the most complex one to configure and use. Our experience is in line with previous usability studies [23].

As one step towards making sandboxing more approachable, the minijail project attempts to unify various sandbox technologies, providing a process-wrapper tool and a self-containment API. Unfortunately, in our experiments we found that the libminijail is currently not particularly well-documented, and using the API requires studying the source code and examples. A similar unifying approach has been taken in academia by Abbadini et al., who proposed multi-technology sandboxing frameworks for JavaScript and TypeScript programs [55], [56]. One of the strengths of their work is a simple JSON-based policy language that allows configuring all aspects of the sandbox. In contrast, minijail currently provides a text-based policy language only for configuring seccomp. Still, we believe these steps towards unification to be a promising development. Hopefully they will make sandboxing usable even to developers who are not security experts.

One interesting and neglected research direction is the combination of sandboxing and remote attestation [57]. In confidential computing [58], remote attestation is used to verify that a a process running on a cloud server is sufficiently well-protected so that attackers cannot, for example, extract sensitive data from it. The process is assumed to be trustworthy after its identity and the security of its platform have been established. In contrast, sandboxing assumes the process will – either by design or due to a compromise – try to harm the system and other processes running on the system. At first glance, it would seem that using remote attestation to verify properties of a sandbox is not useful. But sandboxing can prevent unpredictable attacks – such as unknown vulnerabilities or backdoors. Thus including a proof of the sandboxing policy and enforcement in attestation evidence would be a useful future step. Sandboxing *within a TEE* is also thus a far mostly unexplored area.

References

[1] C. Greamo and A. Ghosh, “Sandboxing and virtualization: Modern tools for combating malware,” *IEEE Security & Privacy*, vol. 9, pp. 79–82, Mar. 2011.

[2] “Zerodium exploit acquisition program,” https://zerodium.com/program.html, 2024.

[3] S. Laurén, S. Rauti, and V. Leppänen, “A survey on application sandboxing techniques,” in *Proceedings of the 18th International Conference on Computer Systems and Technologies*, ser. CompSysTech ’17. New York, NY, USA: Association for Computer Machinery, 2017, pp. 141–148.

[4] J. Anderson, “A comparison of Unix sandboxing techniques,” *FreeBSD Journal*, pp. 8–12, Sep. 2017.

[5] R. Shu, P. Wang, S. A. G. III, B. Andow, A. Nadkarni, L. Deshotels, J. Gionta, W. Enck, and X. Gu, “A study of security isolation techniques,” *ACM Computing Surveys*, vol. 49, Oct. 2016.

[6] T. Dunlap, W. Enck, and B. Reaves, “A study of application sandbox profiles in Linux,” in *SACMAT’22: Proceedings of the 27th ACM Symposium on Access Control Models and Technologies*. New York, NY, USA: Association for Computer Machinery, Jun. 2022, pp. 19–30.

[7] R. Mayrhofer, J. V. Stoep, C. Brubaker, D. Hackborn, B. Bonné, G. S. Tuncay, R. P. Jover, and M. A. Specter, “The Android platform security model (2023),” *arXiv*, Dec. 2023.

[8] J. Lim, Y. Jin, M. Alharthi, X. Zhang, J. Jung, R. Gupta, K. Li, D. Jang, and T. Kim, “SoK: On the analysis of web browser security,” *CoRR*, vol. abs/2112.15561, 2021, arXiv pre-print.

[9] H. C. A. van Tilborg and S. J. (Eds.), *Encyclopedia of Cryptography and Security*. Heidelberg: Springer Verlag, 2011.

[10] M. Bursell, *Trust in Computer Systems and the Cloud*. Hoboken, New Jersey, USA: John Wiley & Sons, 2022.

[11] L. Gunn, N. Asokan, J.-E. Ekberg, H. Liljestrand, V. Nayani, and T. Nyman, “Hardware platform security for mobile devices,” *Foundations and Trends in Privacy and Security*, vol. 3, pp. 214–394, Jun. 2022.

[12] E. Bugnion, J. Nieh, and D. Tsafrir, *Hardware and software support for virtualization*, ser. Synthesis Lectures on Computer Architecture. Kentfield, CA, USA: Morgan & Claypool Publishers, 2017.

[13] L. Rice, *Container Security*. Sebastobol, CA, USA: O’Reilly Media, Inc., 2020.

[14] R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom, “The use of name spaces in Plan 9,” in *Proceedings of the 5th Workshop on ACM SIGOPS European Workshop: Models and Paradigms for Distributed Systems Structuring*. New York, NY, USA: Association for Computer Machinery, 1992, pp. 1–5.

[15] A. Viro and R. Pai, “Shared-subtree concept, implementation and applications in Linux,” in *Proceedings of the 2006 Ottawa Linux Symposium*, Jun. 2006.

[16] E. W. Biedermann, “Multiple instances of global Linux namespaces,” in *Proceedings of the Linux Symposium*, 2006, pp. 101–112.

[17] D. M. Ritchie and K. Thompson, “The UNIX time-sharing system,” *Communications of the ACM*, vol. 17, pp. 365–375, Jul. 1974.

[18] J. L. Obes, “Minijail: Running untrusted programs safely,” in *Linux Security Summit 2016*. Linux Foundation, 2016.

[19] T. Garfinkel, “Traps and pitfalls: Practical problems in system call interposition based security tools,” in *Proceedings of the Network and Distributed Systems Security Symposium (NDSS 2003)*, vol. 3. Internet Society, 2003, pp. 163–176.

[20] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman, “Linux security modules: General security support for the Linux kernel,” in *Proceedings of the 11th USENIX Security Symposium*. Boston, MA, USA: The USENIX Association, 2002.

[21] R. Watson, B. Feldman, A. Migus, and C. Vance, “Design and implementation of the TrustedBSD MAC framework,” in *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX’03)*. IEEE Computer Society, 2003, pp. 38–49.

[22] M. Salaun, “Landlock LSM: Toward unprivileged sandboxing,” https://landlock.io/talks/2017-09-14_landlock-lss.pdf, 2017.

[23] Z. C. Schreuders, T. J. McGill, and C. Payne, “Towards usable application-oriented access controls: Qualitative results from a usability study of SELinux, AppArmor and FBAC-LSM,” *International Journal of Information Security and Privacy*, vol. 6, pp. 57–76, Jan. 2012.

[24] P.-H. Kamp and R. N. M. Watson, “Jails: Confining the omnipotent root,” in *Proceedings of the 2nd International SANE Conference*, vol. 43, May 2000.

[25] R. E. Smith, “Mandatory protection for Internet server software,” in *Proceedings of the 12th Annual Computer Security Applications Conference*. IEEE, 1996, pp. 178–184.

[26] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer, “A secure environment for untrusted helper applications (confining the wily hacker),” in *Proceedings of the Sixth USENIX UNIX Security Symposium*, Jul. 1996.

[27] D. A. Wagner, “Janus: an approch for confinement of untrusted applications,” Master’s thesis, University of California, Berkeley, 1999.

[28] N. Provos, M. Friedl, and P. Honeyman, “Preventing privilege escalation,” in *12th USENIX Security Symposium*. Boston, MA, USA: USENIX Association, Aug. 2003.

[29] C. Miller, D. Blazakis, D. DaiZovi, S. Esser, V. Iozzo, and R.-P. Weinmann, *iOS Hacker’s Handbook*. Indianapolis, IN, USA: John Wiley & Sons, Apr. 2012.

[30] M. Blochberger, J. Rieck, C. Burkert, T. Mueller, and H. Federrath, “State of the sandbox: Investigating macOS application security,” in *Proceedings of the 18th ACM Workshop on Privacy in Electronic Society (WPES ’19)*. New York, NY, USA: Association for Computer Machinery, Nov. 2019, pp. 150–161.

[31] E. Benoist-Vanderbeken and F. Perigaud, “WEN ETA JB? a 2 million dollars problem,” in *SSTIC19: Symposium sur la sécurité des technologies de l’information et des communications*. Cesson-Sévigné, France: Association STIC, Jun. 2019.

[32] ——, “An Apple a day keeps the exploiter away,” in *SSTIC22: Symposium sur la sécurité des technologies de l’information et des communications*. Cesson-Sévigné, France: Association STIC, Jun. 2022.

[33] D. Damopoulos, G. Kambourakis, and S. Gritzalis, “iSAM: An iPhone stealth airborne malware,” in *Future Challenges in Security and Privacy for Academia and Industry*. Berlin and Heidelberg, Germany: Springer Heidelberg Berlin, 2011, pp. 17–28.

[34] C. Spensky, J. Stewart, A. Yerukhimovich, R. Shay, A. Trachtenberg, R. Housley, and R. K. Cunningham, “SoK: Privacy on mobile devices - it’s complicated,” *Proceedings on Privacy Enhancing Technologies*, vol. 2016, no. 3, pp. 96–116, 2016.

[35] R. Deaconescu, L. Deshotels, M. Bucicoiu, W. Enck, L. Davi, and A.-R. Sadeghi, “SandBlaster: Reversing the Apple sandbox,” *arXiv pre-print 1608.04303*, 2016.

[36] C. Fitzl, “Mitigating exploits using Apple’s endpoint security,” in *Virus Bulletin Conference (VB2021 localhost)*. Oxfordshire, UK: Virus Bulletin Ltd, Oct. 2021.

[37] D. Blazakis, “The Apple sandbox,” in *Black Hat USA*, Jan. 2011.

[38] L. Deshotels, R. Deaconescu, M. Chiroiu, L. Davi, W. Enck, and A.-R. Sadeghi, “SandScout: Automatic detection of flaws in iOS sandbox profiles,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 704–716.

[39] J. Rieck, “A whirlwind tour of the Apple sandbox,” https://ubrigens.com/posts/sandbox_tour.html, Feb. 2020.

[40] R. Mayrhofer, J. V. Stoep, C. Brubaker, and N. Kralevich, “The Android platform security model,” *ACM Transactions on Privacy and Security*, vol. 24, Apr. 2021.

[41] “Appcontainer for legacy apps,” https://learn.microsoft.com/en-us/windows/win32/secauthz/appcontainer-for-legacy-applications-, Microsoft, 2023.

[42] P. Yosifovich, D. A. Solomon, and A. Ionescu, *Windows Internals, Part 1: System architecture, processes, threads, memory management and more*. Microsoft Press, 2017.

[43] “Appcontainer isolation,” https://learn.microsoft.com/en-us/windows/win32/secauthz/appcontainer-isolation, Microsoft, 2023.

[44] “App capability declarations,” https://learn.microsoft.com/en-us/windows/uwp/packaging/app-capability-declarations, Microsoft, 2023.

[45] M. Kerrisk, “Using seccomp to limit the kernel attack surface,” in *Linux Plumbers Conference 2015*, 2015.

[46] J. Jia, Y. Zhu, D. Williams, A. Arcangeli, C. Canella, H. Franke, T. Feldman-Fitzthum, D. Skarlatos, D. Gruss, and T. Xu, “Programmable system call security with eBPF,” https://arxiv.org/abs/2302.10366, 2023, arXiv:2302.10366.

[47] C. Negus, *Linux Bible*, 9th ed. Indianapolis, IN, USA: John Wiley & Sons, 2015.

[48] N. Elenkov, *Android Security Internals*. San Fransisco, CA, USA: No Starch Press, Inc, 2015.

[49] T. Harada, T. Horie, and K. Tanaka, “Task oriented management obviates your onus on Linux,” in *Linux Conference*, 2004.

[50] M. Bauer, “Paranoid penguin - AppArmor on Ubuntu 9,” *Linux Journal*, 2009.

[51] A. Gruenbacher and S. Arnold, “AppArmor technical documentation,” https://lkml.iu.edu/hypermail/linux/kernel/0706.1/0805/techdoc.pdf, SUSE Labs / Novell, Apr. 2007.

[52] “aa-exec - confine a program with the specified apparmor profile,” Canonical Ltd.

[53] “minijail,” https://google.github.io/minijail/, 2024.

[54] “minijail0(1): sandbox a process,” https://google.github.io/minijail/minijail0.1.html, 2024.

[55] M. Abbadini, D. Facchinetti, G. Oldani, M. Rossi, and S. Paraboschi, “Cage4Deno: A fine-grained sandbox for Deno subprocesses,” in *ACM ASIA Conference on Computer and Communications Security (ASIA CCS ’23)*. New York, NY, USA: Association for Computer Machinery, Jul. 2023.

[56] ——, “NatiSand: Native code sandboxing for JavaScript runtimes,” in *The 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID ’23)*. New York, NY, USA: Association for Computer Machinery, Oct. 2023, pp. 639–653.

[57] A. Niemi, S. Sovio, and J.-E. Ekberg, “Towards interoperable enclave attestation: Learnings from decades of academic work,” in *2022 31st Conference of Open Innovations Association (FRUCT)*. IEEE, Apr. 2022, pp. 189–200.

[58] M. Russinovich, “Confidential Computing: Elevating cloud security and privacy,” *Communications of the ACM*, vol. 67, pp. 52–53.