gVisor Report

Intro (summary, core domain)

gVisor is a container sandboxer that aims to provide highly isolated environments in which to execute less than fully trusted applications while solving a multitude of problems and inaptitudes inherent to containers and sandboxing with virtual machines.

In recognition of the gaps in isolation resulting from all containers sharing a single host operating system and all VMs sharing a hypervisor, gVisor provides a user-level implementation of Linux - the Sentry. This user-level application provides the system API to containerized applications while preventing any untrusted application from directly interacting with the host OS. Providing each sandbox with its own independent Sentry prevents untrusted applications from escaping their isolated environment through direct attacks on the shared hypervisor that are possible with traditional sandboxes. While the Sentry implements the Linux system API, the API exposed to guest applications is not identical to that of Linux. Confines are imposed on the set of system calls available to both guest applications and the Sentry itself in order to lessen the attack surface on the system. The attack suface is further diminished by relocating file system operations such as `open()` to a seperate module Gofer, independent and isolated from the Sentry.

In addition to its security and isolation goals, gVisor is designed with the intent to minimize start up time for containers. This is evidenced by its main deviation from traditional container sandboxing with virtual machines - rather than creating virtualized hardware for a guest kernel that guest applications will interact with, the system API provided by the Sentry is based directly off the host system API with the Sentry deferring to the host at runtime for some services.

In relation to alternatives to gVisor for isolating less than fully trusted applications for execution on a shared system such as traditional linux containers, gVisor suffers from dreadful performance. The performance hits to gVisor directly result from the exact features of the system that provide the security, isolation, and defense in depth it was built for. While the performance of gVisor lacks in relation to traditional containers, it is optimized for efficient building and destroying of containers and achieves better performance in this than container sandboxes implemented via virtualized hardware and a guest kernel.

The poor performance of gVisor in memory usage, networking, I/O and more make this system not well suited for extensive applications that execute for longer periods of time and necessitate abuntant resources. gVisor is best suited for serverless frameworks leveraging Function as a Service where applications will be run for short periods of time to provide a service, then are destroyed. The ability of gVisor to quickly build and destroy containers furthers the suitability of the system for serverless frameworks as this infrastructure necessitates multiple instantiations of these containers to execute services for many clients. Clearly minimizing construction and deconstruction time of containers maximizes the effiency and productivity of the service provider.

Function as a Service has become more prevelent in recent years and cloud computing has grown - gVisor provides the requisite security principles and depth for the execution of untrusted guest applications from a myriad of clients in cloud computation, and is optimized for the Function as a Service model frequently implemented in modern cloud computing. As a result of these core features of gVisor, it is highly suitable and immensely valuable in cloud computing while the performance tradeoffs of these core features make gVisor a poor fit for just about any other purpose or environment.

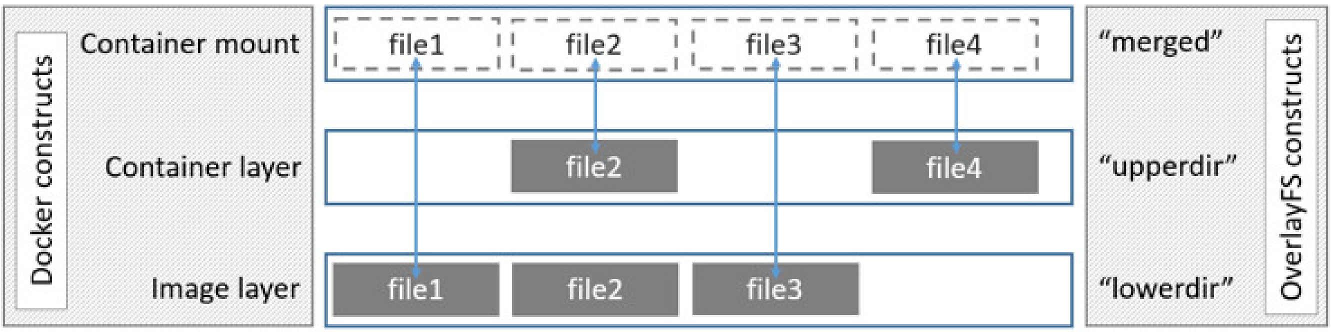
Modules (Sentry, Gofer, runsc, platforms)

Gofer

Filesystem isolation boundaries

The Gofer module is responsible for maintaining isolation boundaries for a sandboxed container's filesystem resources. Gofer is a file proxy that runs as a separate process, isolated from the sandbox, mediating access to filesystem resources. A separate gofer instance runs for each running sandbox instance. They communicate with their respective Sentrys using the 9p protocol. The host filesystem is isolated from the sandbox using an overlay file system. This creates a temporary union mount filesystem. Thus all changes made to files are stored within the sandbox but do not affect the host file system.

Diagram of overlayfs:



The above image shows three layers. The bottom layer is called the `lowerdir` and represents the base image the container will mount. The `upperdir` holds the container's bundled file directory with all of its code. The container is mounted on `merged` which is the overlay layer. `merged` appears to be a full image/filesystem with all of its own code to the container. The `lowerdir` layer is readonly and changes here will not change the actualy base image stored on the host. Changing or adding files to the `lowerdir` will create a copy of the file in the `upperdir` and changes are stored there. Deletions are implemented with whiteout files/directories that are transparently stored in the overlay layer. Changes to the `upperdir` can take place as normal. This copy-on-write feature is useful because the container can use the image at will without having to copy the entire directory into a new sandboxed filesystem. Sources: paper discussing overlayfs in containers, kernel docs

```
// line 39 in gvisor/runsc/cmd/gofer.go
type Gofer struct {
    bundleDir string   //refers to the directory containing the executable code
    ioFDs     intFlags //file descriptors used to communicate with 9p servers
    applyCaps bool     //boolean var sets whether or not capabilities are
                       //used to restrict the Gofer process. Default = true
    setUpRoot bool     //boolean var indicates whether or not an empty root should
                       //set up for the process. Default = true
    specFD   int       //file descriptor pointing to the OCI runtime spec file
    mountsFD int       //mountsFD is the file descriptor to write list of mounts
                       //after they have been resolved (direct paths, no symlinks)
}
```

Gofer is responsible for mounting this filesystem and controlling access. As shown above, important variables tracked in the Gofer struct help with this implementation. The runtime spec file referred to by `specFD` (example here) is used to set up Gofer's mount namespace. Gofer first turns all shared mounts into slave mounts so that changes can propagate into the shared mounts but not outside of the namespace into the host. Next, the root needs to be mounted on a new `tmpfs` (temporary filesystem). `runsc` requires a `/proc` directory so the new `tmpfs` is mounted here. Under `/proc`, new directories `/proc/proc` and `/proc/root` are created to give a location for the new sandboxed root. The new `/proc/proc` is mounted with the following flags for to prevent any attempts to break out of the isolated sandbox:

- MS_RDONLY
  - don't allow process to write changes.
- MS_NOSUID
  - don't allow system to use/contain set user id files. This helps to prevent privilege escalation.
- MS_NODEV
  - don't allow access to devices or special files on the filesystem
- MS_NOEXEC
  - don't allow program execution on this filesystem

The container's source directory specified by the spec file is then mounted on the new `/proc/root`. The following flags are used:

- MS_BIND
  - Bind mount takes an existing filesystem structure and places it in a new location in the file system. This is used to place the Gofer process's directory, which is saved on the host, in an isolated location within the sandbox without copying everything over.
- MS_SLAVE
  - Turns a shared mount point into a slave mount point. This means changes can propagate from the source (the host) but changes made in the slave will not propagate to the source. Any attempts to maliciously delete files will only effect the current running sandboxed process.
- MS_REC
  - Recursively propagate these mount options to all subdirectories of the mount point.

```
// line 39 in gvisor/runsc/cmd/chroot.go
// pivot_root(".", ".") makes a mount of the working directory the new
// root filesystem, so it will be moved in "/" and then the old_root
// will be moved to "/" too. The parent mount of the old_root will be
// new_root, so after umounting the old_root, we will see only
// the new_root in "/".
if err := unix.PivotRoot(".", "."); err != nil {
    return fmt.Errorf("pivot_root failed, make sure that the root mount has a
parent: %v", err)
}
if err := unix.Unmount(".", unix.MNT_DETACH); err != nil {
    return fmt.Errorf("error umounting the old root file system: %v", err)
}
```

Source: gvisor/runsc/cmd/chroot.go

After setting up the root of the filesystem, the rest of any subsequent mounts necessary for the container's execution are mounted in the correct location under the new root. The next step is to use change directories to the new `/proc` and call `pivotRoot()`. This changes the root of the Gofer process's filesystem namespace and then unmounts the old root. Lastly, Gofer uses `chroot` to further isolate the processes root one more subdirectory to `/root` (because `/proc/root` just turned into `/root` in the process's namespace). Now the process sees `/root` as `/`, it's own image, containing all code needed to execute, completely isolated from the host OS.

Process limitations

After setting up the filesystem, Gofer uses two methods of limiting what the process can do. It limits Capabilities and system calls. First it applies the minimal set of capabilities required to operate on files:

```
// line 39 in gvisor/runsc/cmd/gofer.go
var caps = []string{
    "CAP_CHOWN",
    "CAP_DAC_OVERRIDE",
    "CAP_DAC_READ_SEARCH",
    "CAP_FOWNER",
    "CAP_FSETID",
    "CAP_SYS_CHROOT",
}
```

Source: gvisor/runsc/cmd/gofer.go

Next it uses seccomp and BPF filters to limit the system calls available to the process. Similarly to the applied capabilities, Gofer whitelists only the system calls required for execution, thus limiting the attack surface for the process.

Server Model

Gofer acts as the file server for the running gVisor containers. This means each container is a client that must request request Gofer to perform any necessary filesystem operations. After sandboxing the Gofer process, a list of p9 attachers is allocated. Then, beginning with `/`, an attach point is created for every mount point specified in the spec file. As well as holding the mount points necessary for the container, the OCI specfile holds a list of `ioFD`s which are file descriptors used to connect to 9P servers. Iterating through the list of associated `ioFD`s and attachment points, a goroutine is started to create a new socket for each `ioFD` and a new p9 server for each attachment point. This logic is shown below.

```
// line 218 in gvisor/runsc/cmd/gofer.go
func runServers(ats []p9.Attacher, ioFDs []int) {
    // Run the loops and wait for all to exit.
    var wg sync.WaitGroup
    for i, ioFD := range ioFDs {
        wg.Add(1)
        go func(ioFD int, at p9.Attacher) {
            socket, err := unet.NewSocket(ioFD)
            if err != nil {
```

```
                Fatalf("creating server on FD %d: %v", ioFD, err)
            }
            s := p9.NewServer(at)
            if err := s.Handle(socket); err != nil {
                Fatalf("P9 server returned error. Gofer is shutting down. FD: %d,
err: %v", ioFD, err)
            }
            wg.Done()
        }(ioFD, ats[i])
    }
    wg.Wait()
    log.Infof("All 9P servers exited.")
}
```

Source: gvisor/runsc/cmd/gofer.go

As shown above, to begin, the gofer creates one goroutine per client connection and calls `Handle()`. `Handle()` creates a connection state for the given connection and calls `handleRequests()`. This method is a simple wrapper that just infinitely calls `handleRequest()` until an error is thrown or the goroutine receives the shutdown signal (i.e. another goroutine detected a connection problem). `handleRequest()` "handles" the main logic of dealing with client connections. The first step is to receive the client request.

```
// Line 121 - ReadVec() in gvisor/pkg/unet/unet_unsafe.go
for {
    var e unix.Errno
    // Try a non-blocking recv first, so we don't give up the go runtime M.
    n, _, e = unix.RawSyscall(unix.SYS_RECVMSG, uintptr(fd),
uintptr(unsafe.Pointer(&msg)), unix.MSG_DONTWAIT|unix.MSG_TRUNC)
    if e == 0 {
        break
    }
    ... error handling ...
    // Wait for the socket to become readable.
    err := r.socket.wait(false)
    ... error handling ...
}
```

Source: gvisor/pkg/unet/unet_unsafe.go

Gofer will attempt to use a non-blocking receive first, and will only block if there is no data incoming. This allows the runtime to be given up when the client isn't currently requesting anything. When there is more traffic on the stream however, the non-blocking receive will handle client requests immediately. This all takes place within a critical section which is entered by taking the receive mutex on the client connection struct, as shown below:

```
// line 518 in gvisor/pkg/p9/server.go
atomic.AddInt32(&cs.recvIdle, 1)
```

```
cs.recvMu.Lock()
atomic.AddInt32(&cs.recvIdle, -1)
```

Before actually handling the message just received, Gofer will make sure another goroutine is ready to handle the next incomming request:

```
// line 545 in gvisor/pkg/p9/server.go
// Ensure that another goroutine is available to receive from cs.conn.
if atomic.LoadInt32(&cs.recvIdle) == 0 {
    go cs.handleRequests() // S/R-SAFE: Irrelevant.
}
cs.recvMu.Unlock()
```

Source: gvisor/pkg/p9/server.go

If `recvIdle` is zero, that means that no goroutines on the given client state (`cs`) are waiting for the mutex. So Gofer spawns another goroutine on the same `cs` to recieve the next message while it handles the current one. If there is one waiting for the mutex, it will be able to receive the next message as soon as the current one releases the mutex. Assuming the goroutines can quickly process and send responses to the clients, this thread pool will never grow too large and will simply reuse the same small set of goroutines per connection.

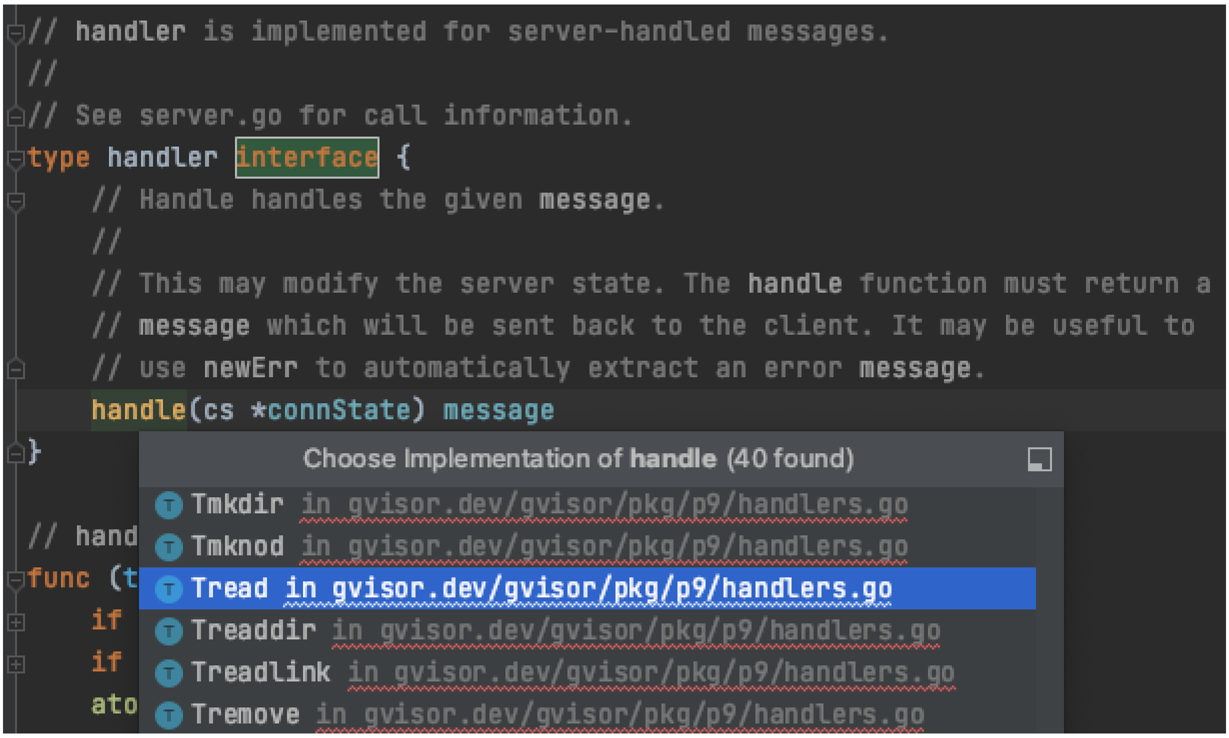
Handling the request and performing an operation.

```
//line 571 in gvisor/pkg/p9/server.go
// Handle the message.
r := cs.handle(m)
...
// Send back the result.
cs.sendMu.Lock()
err = send(cs.conn, tag, r)
cs.sendMu.Unlock()
```

Source: gvisor/pkg/p9/server.go

To actually handle the request and perform some filesystem operation, `cs.handle()` is called with the received message. The action depends on the message, so the message handler function implements the handler interface shown below to carry out the appropriate action. Some different options are also shown below.

line 68 in [gvisor/pkg/p9/handlers.go]



For example, a `Tread` request carries out a read request and returns the data requested. A `Twrite` request will carry out a write and return the number of bytes successfully written. Whatever is returned in `r` is then sent back to the client to complete this interaction.

Platforms

The platform module of gVisor is essentially the Virtual Machine Monitor for the system. The platform handles context switches and the mapping of memory as well as the intercepting of system calls from guest applications running in a virtual machine. gVisor offers two implementations of the platform model - ptrace and KVM. While both platform implementations support the same functionalities there are distinct differences and clear tradeoffs between them. The ptrace implementation offers higher portability as it can run anywhere ptrace works while KVM only functions on bare hardware or in VMs with nested virtualization enabled. The lesser constraints on deployment of ptrace provides more widespread compatability for gVisor, but at a cost. The ptrace implementation has a far higher overhead for context switches than KVM and therefore ill-suited for deployment on systems with a high rate of system call-heavy guest applications. Source: [gVisor platform guide]

This difference in context switch overhead is a result of the distinct ways in which the platform implementations handle the interception and forwarding of system calls.

In the KVM implementation the bluepill handler runs on an infinite `for{}`, calling `KVM_RUN` each loop. This handler is switched to when a context switch out of the guest application level occurs, for instance due to a system call or page fault. The handler checks the reason for the exit from guest user and directly forwards the request to the Sentry in Ring0. Once the exit reason is determined `bluepill_handler` invokes the appropriate bluepill functions to exit guest user and service the request or fault that caused the context switch.

```
for {
        _, _, errno := unix.RawSyscall(unix.SYS_IOCTL, uintptr(c.fd), _KVM_RUN, 0)
```

```
// escapes: no.
	switch errno {
	case 0: // Expected case.
```

Source: /gvisor/pkg/sentry/platform/kvm/bluepill_unsafe.go

In contrast to KVM forwarding system calls to the Sentry in Ring0 through the invokation of bluepill functions, the ptrace implementation instead handles system call forwarding by making the same system call received from the guest application with PTRACE enabled in order to prevent the host kernel from actually servicing the request as Ptrace in the host kernel forwards the system call to the Sentry.

```
func (t *thread) syscall(regs *arch.Registers) (uintptr, error) {
        // Set registers.
    if err := t.setRegs(regs); err != nil {
        panic(fmt.Sprintf("ptrace set regs failed: %v", err))
    }

    for {
        // Execute the syscall instruction. The task has to stop on the
        // trap instruction which is right after the syscall
        // instruction.
        if _, _, errno := unix.RawSyscall6(unix.SYS_PTRACE, unix.PTRACE_CONT,
uintptr(t.tid), 0, 0, 0, 0); errno != 0 {
            panic(fmt.Sprintf("ptrace syscall-enter failed: %v", errno))
        }
```

Source: /gvisor/pkg/sentry/platform/ptrace/subprocess.go

While this maintains the same isolation and principles of defense in depth of the KVM implementation, this implementation of system call handling results in a definitively larger overhead for ptrace mode. However, this extra redirection and utilization of Ptrace in the host kernel for redirection is precisely what gives the Ptrace platform its superior compatability properties compared to KVM. Since KVM directly forwards system calls to the Sentry it cannot run in a virtual machine with nested virtualization disabled. Ptrace, however, can run inside of a VM with virtualization disabled as the system calls will simply be made to the VM hypervisor in Ptrace mode where they will then be redirected to the Sentry - this implementation eliminates the need for a hypervisor (such as KVM) executing within a hypervisor.

In addition to handling context switches and system call forwarding, the platform is responsible for memory mappings between both guest applications and the Sentry as well as the initialization of memory reserved to and managed by the Sentry. When a Sentry is built and the guest physical memory for said Sentrys sandbox is allocated the platform makes the `mmap()` system call to the host kernel to fill the host address space with `PROT_NONE` mappings to set up the guest physical memory of the Sentry.

```
// physicalInit initializes physical address mappings.
func physicalInit() {
physicalRegions = computePhysicalRegions(fillAddressSpace())
}
```

```
for filled := uintptr(0); filled < required && current > 0; {
addr, _, errno := unix.RawSyscall6(
unix.SYS_MMAP,
0, // Suggested address.
current,
unix.PROT_NONE,
unix.MAP_ANONYMOUS|unix.MAP_PRIVATE|unix.MAP_NORESERVE,
0, 0)
```

Source: [/gvisor/pkg/sentry/platform/kvm/physical_map.go]

When a new sandbox is created the platform creates a new page table and computes mappings from the sandboxes guest virtual addresses to Sentrys guest physical regions. Throughout execution the platform performs translations between applications guest virtual addresses and the Sentrys guest physical addresses.

```
// PhysicalFor returns the physical address for a set of PTEs.
//
// +checkescape:all
//
//go:nosplit
func (a *allocator) PhysicalFor(ptes *pagetables.PTEs) uintptr {
```

```
func (a *allocator) LookupPTEs(physical uintptr) *pagetables.PTEs {
```

Source: [/gvisor/pkg/sentry/platform/kvm/physical_map.go]

While both KVM and Ptrace serve alongside the Sentry as the virtual machine monitor, their representations of the guest applications they manage differs. The difference in how each platform implementation represent and track data for guest applications is likely due to the added requirements of Ptrace in order to properly trace the execution and state of said guest application.

In the KVM implementation, guest applications are simply represented as a `context` which consists of an address space, register set, a `machine` and a few other bookkeeping data items. The `machine` data structure is where KVM maintains data on a particular VM. A switch simply consists of enabling interrupts on the virtual CPU of the machine, setting the address space as active, and loading register states.

While the Ptrace implementation of context switches also maintains similar data to KVM in a `context` data structure, the tracing of guest applications necessitates a subprocess to trace threads. Host threads are created depending on the number of active guest applications within a sandbox. A subprocess is a collection of traced threads, consisting of a pool of threads reserved for emulation, one reserved for system calls. The need to trace the execution of threads necessitates these subprocesses and complicates context switches in this platform. In order to switch to a given context the current runtime thread must be locked, then Ptrace must find the traced subprocess for this runtime thread and then perform the operation in this traced subprocess.

Source: [/gvisor/pkg/sentry/platform/ptrace/ptrace.go]

While the Ptrace implementation of the platform offers more general compatability than KVM, the necessities of tracing thread execution complicate the operations supported by the platform. In addition to the added complexity of having traced subprocesses backing guest threads, the way in which Ptrace handles system call redirection to the Sentry results in higher overhead than KVM, having the effect of decreasing the practical applications of Ptrace-mode gVisor.

Sentry

The Sentry is the largest component of gVisor. It acts an a kernel to any application running in the sandbox. When an application running in the sandbox makes a system call, that system call is first intercepted by the Platform, then passed to the Sentry. Using a limited set of 55 host system calls, the Sentry implements 211 system calls for sandboxed applications. System calls are never passed directly from a sandboxed application to the host kernel. If possible, the Sentry will handle the call without making any system calls to the host. For system calls related to filesystem access, the Sentry makes requests Gofer using the 9P protocol. Other than system calls made to the host by Sentry, the sandbox runs exclusively in user level.

Abstractions (and code examples showing implementation with type defs)

Threads and Processes

Because gVisor runs its own kernel via the Sentry, a gVisor sandbox appears as a single process to the host system, regardless of how many processes are running within the sandbox. The Sentry creates a `Task` struct for each thread of execution within the sandbox. These `Task`s are dispatched as a `goroutine`, a many-to-one user-space thread model provided by the Go language and can be bundled together into a `TaskSet` to support multithreaded applications. `Task`s are scheduled by the Sentry and unknown to the host. The Sentry can create host threads as needed to support a varying volume of `Task`s running in the sandbox. A subset of the `Task` struct definition can be seen below. Similar to a `proc`, it includes fields for state, size of memory, scheduling, mutexes, filesystem context, priority, and cpu assignment. Source: [gvisor/pkg/sentry/kernel/task.go]

```
type Task struct {
    taskNode

    //task goroutine's ID
    goid int64 `state:"nosave"`

    // runState is what the task goroutine is executing if it is not stopped.
    // If runState is nil, the task goroutine should exit or has exited.
    // runState is exclusive to the task goroutine.
    runState taskRunState

    // current scheduling state of the task goroutine.
    // gosched is protected by goschedSeq. gosched is owned by the task goroutine.
    goschedSeq sync.SeqCount `state:"nosave"`
    gosched    TaskGoroutineSchedInfo
```

```
    // p provides the mechanism by which the task runs code in userspace. The p
    // interface object is immutable.
    p platform.Context `state:"nosave"`

    // k is the Kernel that this task belongs to. The k pointer is immutable.
    k *Kernel

    // mu protects some of the following fields.
    mu sync.Mutex `state:"nosave"`

    // fsContext is the task's filesystem context.
    // fsContext is protected by mu, and is owned by the task goroutine.
    fsContext *FSContext

    // fdTable is the task's file descriptor table.
    // fdTable is protected by mu, and is owned by the task goroutine.
    fdTable *FDTable

    // ipcns is the task's IPC namespace.
    // ipcns is protected by mu. ipcns is owned by the task goroutine.
    ipcns *IPCNamespace

    // cpu is the fake cpu number returned by getcpu(2). cpu is ignored
    // entirely if Kernel.useHostCores is true.
    // cpu is accessed using atomic memory operations.
    cpu int32

    // This is used to keep track of changes made to a process' priority/niceness.
    niceness int

    // startTime is the real time at which the task started.
    // startTime is protected by mu.
    startTime ktime.Time
```

`Task` objects can be grouped together into a `TaskSet` for running multi-threaded applications within the sandbox. A `TaskSet` includes a `PIDNamespace` containing a map of `Task`s and various mechanisms for managing concurrent execution of those `Task`s. Source: gvisor/pkg/sentry/kernel/threads.go

```
type TaskSet struct {
    // mu protects all relationships between tasks and thread groups in the
    // TaskSet. (mu is approximately equivalent to Linux's tasklist_lock.)
    mu sync.RWMutex `state:"nosave"`

    // Root is the root PID namespace, in which all tasks in the TaskSet are
    // visible. The Root pointer is immutable.
    Root *PIDNamespace

    // sessions is the set of all sessions.
    sessions sessionList

    // stopCount is the number of active external stops applicable to all tasks
```

```
    // in the TaskSet (calls to TaskSet.BeginExternalStop that have not been
    // paired with a call to TaskSet.EndExternalStop). stopCount is protected
    // by mu.
    //
    // stopCount is not saved for the same reason as Task.stopCount; it is
    // always reset to zero after restore.
    stopCount int32 `state:"nosave"`

    // liveGoroutines is the number of non-exited task goroutines in the
    // TaskSet.
    //
    // liveGoroutines is not saved; it is reset as task goroutines are
    // restarted by Task.Start.
    liveGoroutines sync.WaitGroup `state:"nosave"`

    // runningGoroutines is the number of running task goroutines in the
    // TaskSet.
    //
    // runningGoroutines is not saved; its counter value is required to be zero
    // at time of save (but note that this is not necessarily the same thing as
    // sync.WaitGroup's zero value).
    runningGoroutines sync.WaitGroup `state:"nosave"`

    // aioGoroutines is the number of goroutines running async I/O
    // callbacks.
    //
    // aioGoroutines is not saved but is required to be zero at the time of
    // save.
    aioGoroutines sync.WaitGroup `state:"nosave"`
}
```

Files

Files in the sandbox can be backed by multiple implementations. For host-native files (where a file descriptor is available), the Gofer may return a file descriptor to the Sentry via SCM_RIGHTS. Interactions with file descriptors use the same system calls as Linux, but the calls are implemeted in the Sentry and Gofer. Files can also be mapped into an application's address space in the sandbox using gVisor's `Mappable` interface, similar to `mmap` on Linux. Multiple sandboxes can use shared memory by mapping the same file. In addition to interacting with files on the host system, gVisor creates filesystems that exist only within the sandbox, such as a `tmpfs` at `/tmp` or `/dev/shm`. These filesystems count against the sandbox's memory allowance from the host.

gVisor filesystems are implemented similarly to Linux as well. A filesystem in gVisor consists of a tree of reference-counted `Dentry` nodes. Each `Dentry` node maintains a reference to a `DentryImpl`, or Dentry implementation. The `DentryImpl` defines how a specific `Dentry` should be managed. Unlike Linux, `Dentry` nodes in gVisor are not associated with inodes. This is due to communication with Gofer occuring through a 9P api rather than raw block devices. In addition, virtual filesystems within a sandbox would lose track of files from the host if the host were to rename an inode. Because `Dentry` nodes are handled this way, filesystems in gVisor are not responsible for deleting `Dentry` nodes with a reference count of 0. `Dentry` reference counts instead reperesent the extent to which a filesystem requires a certain `Dentry` node. The filesystem can continue to cache `Dentry` nodes with no references, or they may be discarded. The `Dentry` interface definintion can be seen below. Source: [pkg/sentry/vfs/dentry.go]

```
type Dentry struct {
    // mu synchronizes deletion/invalidation and mounting over this Dentry.
    mu sync.Mutex `state:"nosave"`

    // dead is true if the file represented by this Dentry has been deleted (by
    // CommitDeleteDentry or CommitRenameReplaceDentry) or invalidated (by
    // InvalidateDentry). dead is protected by mu.
    dead bool

    // mounts is the number of Mounts for which this Dentry is Mount.point.
    // mounts is accessed using atomic memory operations.
    mounts uint32

    // impl is the DentryImpl associated with this Dentry. impl is immutable.
    // This should be the last field in Dentry.
    impl DentryImpl
}
```

Memory

All memory within a gVisor sandbox is managed by the Sentry using demand-paging and backed by a single `memfs`. Address space creation is platform specific, and for some platforms, the Sentry may create addtional helper processes on the host to support additional address spaces. Like the sandbox itself, the helper processes are subject to various usage limits. Physical memory is all controlled by the host. The Sentry populates mappings from the host and allows the host to control demand-paging. The Sentry will not demand an individual page of memory. Instead, it uses memory-allocation heuristics to select regions. Generally, the Sentry can't tell whether a page is active and only provides approximate usage statistics. It can gather more accurate information if required, but only with an expensive API call. Pages are swapped and reclaimed by the host without the Sentry knowing. Providing that information to the Sentry would open the sandbox too much. The exception to this rule is when an application frees memory. The Sentry immediately releases that memory back to the host, allowing the host to most effectively manage multiplexed resources and its own memory allocation policies. There is potential for this to slow performance in the sandbox; if the Sentry needs that memory again, it must make another request to the host. Lastly, the Sentry also maintains an internal cache for storing files needed in the sandbox that can't be referenced with a host file descriptor.

Security

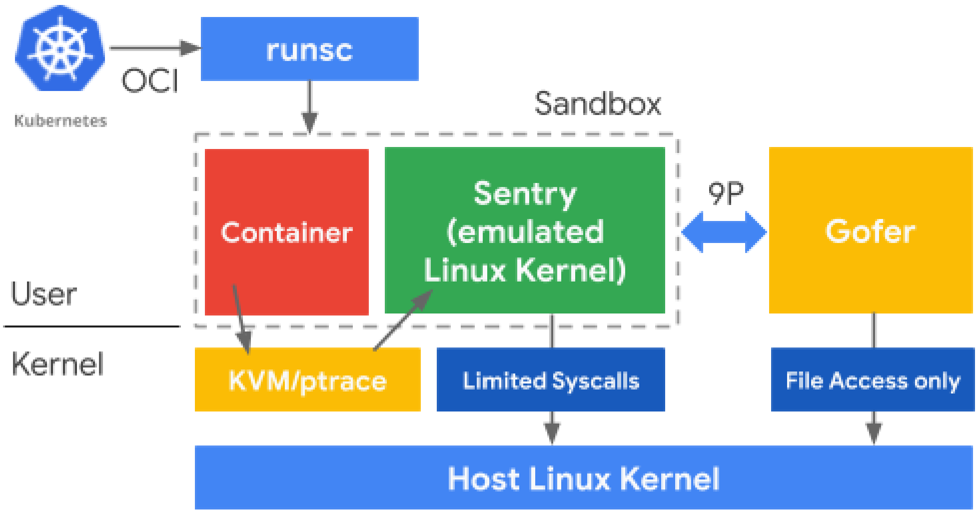
Overview

gVisor’s purpose is to provide a secure container for processes that are assumed to be untrusted – if you did not write it, it is untrusted! Or, at least, that is the philosophy for gVisor. Security in depth allows for the ability to run processes in gVisor without the worry of malicious system call use, troublesome file I/O (thank you, Gofer), and even Linux security vulnerabilities that affect other containers: see [here].

How does gVisor accomplish this level of depth in security? It employs the following mechanisms:

1. Re-routing and intercepting system calls from the untrusted process.
2. Individual implementations of Linux system calls in the Sentry.
3. Reduced set of whitelisted system calls allowed for the Sentry.
4. File operations provided by Gofer over the 9P protocol.

Source



Security Fig. 1

Re-routing and intercepting system calls from the untrusted process.

gVisor ensures that the sandboxed application does not give system calls directly to the host. Threads from the application are tracked, or traced rather, by the Sentry's ptrace implementation. ptrace attaches a tracer to each necessary application thread, as well as all the init() options for the thread's tracking. So, when the application makes a system call, ptrace can redirect over to the Sentry to process the system call, whether that be executing in user level, not allowing the system call at all, or making a call out to the host. In addition to ptrace, the KVM's ring0 platform ensures that system calls from the application are caught in guest mode and handled accordingly by the supplied system call handler and sent over to the Sentry. See platforms for a more detailed overview of KVM.

Source - ptrace

Source - ring0

The security baton is then handed off to the Sentry to process the system call request. If the call can be done completely in user-level with the implemented system calls in the Sentry, it will do so in order to have unnecessary switches out of user-level. If there needs to be a call out to the host, then the Sentry can do so - the reduced set of whitelisted system calls allows for this to be less of an attack surface. Seccomp filters are used for this whitelisting. Lastly, if the system call is not allowed by the Sentry, then it will *not* be performed, but the application will have *no* knowledge of this capability block.

Let us look at some examples.

This function attaches a ptrace to a particular thread for tracing. Additionally, options for the ptrace are initialized.

```
// gvisor/pkg/sentry/platform/ptrace/subprocess.go, LINE 282
```

```
// attach attaches to the thread.
func (t *thread) attach() {
    if _, _, errno := syscall.RawSyscall6(syscall.SYS_PTRACE,
syscall.PTRACE_ATTACH, uintptr(t.tid), 0, 0, 0, 0); errno != 0 {
        panic(fmt.Sprintf("unable to attach: %v", errno))
    }

    // PTRACE_ATTACH sends SIGSTOP, and wakes the tracee if it was already
    // stopped from the SIGSTOP queued by CLONE_PTRACE (see inner loop of
    // newSubprocess), so we always expect to see signal-delivery-stop with
    // SIGSTOP.
    if sig := t.wait(stopped); sig != syscall.SIGSTOP {
        panic(fmt.Sprintf("wait failed: expected SIGSTOP, got %v", sig))
    }

    // Initialize options.
    t.init()
}
```

Code Ex. 1

This code excerpt shows the logic for checking a system call for whether it is being tracked, needs to be run in user-level, or can be invoked.

ptraceSyscallEnter in repo

doSyscallEnter in repo

```
// gvisor/pkg/sentry/kernel/ptrace.go

// ptraceSyscallEnter is called immediately before entering a syscall to check
// if t should enter ptrace syscall-enter-stop.
func (t *Task) ptraceSyscallEnter() (taskRunState, bool) {
    if !t.hasTracer() {
        return nil, false
    }
    t.tg.pidns.owner.mu.RLock()
    defer t.tg.pidns.owner.mu.RUnlock()
    switch t.ptraceSyscallMode {
    case ptraceSyscallNone:
        return nil, false
    case ptraceSyscallIntercept:
        t.Debugf("Entering syscall-enter-stop from PTRACE_SYSCALL") // tracking
the syscall
        t.ptraceSyscallStopLocked()
        return (*runSyscallAfterSyscallEnterStop)(nil), true
    case ptraceSyscallEmu:
        t.Debugf("Entering syscall-enter-stop from PTRACE_SYSEMU") // no host
system calls for you
        t.ptraceSyscallStopLocked()
        return (*runSyscallAfterSysemuStop)(nil), true
```

```
    }
    panic(fmt.Sprintf("Unknown ptraceSyscallMode: %v", t.ptraceSyscallMode))
}

// teleporting...

// gvisor/pkg/sentry/kernel/task_syscall.go

func (t *Task) doSyscallEnter(sysno uintptr, args arch.SyscallArguments)
taskRunState {
    if next, ok := t.ptraceSyscallEnter(); ok { // this means that either a) it
needs to be run in user-level, or b) it is being traced
        return next
    }
    return t.doSyscallInvoke(sysno, args) // otherwise, execute!
}
```

Code Ex. 2

Like mentioned previously, if the thread does not have the proper privilege to perform a capability, then it will not allow the action, but it will still look like there is an implementation for the system call to the callee.

CapError in repo

```
// gvisor/pkg/sentry/syscalls/syscalls.go

// CapError gives a syscall function that checks for capability c.  If the task
// has the capability, it returns ENOSYS, otherwise EPERM. To unprivileged
// tasks, it will seem like there is an implementation.
func CapError(name string, c linux.Capability, note string, urls []string)
kernel.Syscall {
    if note != "" {
        note = note + "; "
    }
    return kernel.Syscall{
        Name: name,
        Fn: func(t *kernel.Task, args arch.SyscallArguments) (uintptr,
*kernel.SyscallControl, error) {
            if !t.HasCapability(c) {
                return 0, nil, syserror.EPERM
            }
            t.Kernel().EmitUnimplementedEvent(t)
            return 0, nil, syserror.ENOSYS
        },
        SupportLevel: kernel.SupportUnimplemented,
        Note:         fmt.Sprintf("%sReturns %q if the process does not have %s;
%q otherwise.", note, syserror.EPERM, c.String(), syserror.ENOSYS),
        URLs:         urls,
    }
}
```

Code Ex. 3

Individual implementations and whitelisting of system calls

The sentry has its own implementation of all whitelisted system calls - the sentry only is allowed a subset of all possible system calls. 51 to be exact, which can all be found in the “gvisor/pkg/sentry/syscalls/linux” directory. This reduced set of system calls allows for a smaller attack surface. The fewer system calls there are, the fewer possibilities there are for an attacker to pass malicious arguments or perform other exploits. This is also assisted by the many layers that are present from when the contained application makes the call, to when the call is invoked, if ever. All of this seems expensive though, doesn't it? Well, it is. gVisor even acknowledges in its documentation that if your contained application needs to make many system calls, there will be a significant performance hit. This is due to the necessity of tracing the system calls and applications for security - without that, the security model is significantly reduced. In KVM mode in contrast with ptrace, there is less overhead but more compatibility issues, so as with most other things, performance is not free.

And here is an example of a whitelisted system call in the Sentry. It is an individual implementation of Linux's pipe(2) system call.

pipe2 in repo

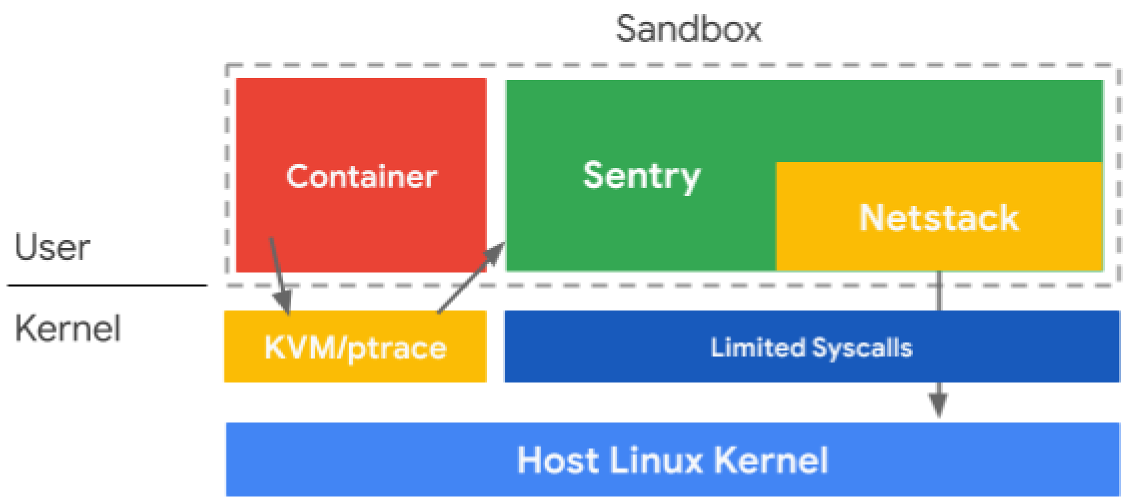
```
// pipe2 implements the actual system call with flags.
func pipe2(t *kernel.Task, addr hostarch.Addr, flags uint) (uintptr, error) {
    if flags&^(linux.O_NONBLOCK|linux.O_CLOEXEC) != 0 {
        return 0, syserror.EINVAL
    }
    r, w := pipe.NewConnectedPipe(t, pipe.DefaultPipeSize)

    r.SetFlags(linuxToFlags(flags).Settable())
    defer r.DecRef(t)

    w.SetFlags(linuxToFlags(flags).Settable())
    defer w.DecRef(t)

    fds, err := t.NewFDs(0, []*fs.File{r, w}, kernel.FDFlags{
        CloseOnExec: flags&linux.O_CLOEXEC != 0,
    })
    if err != nil {
        return 0, err
    }

    if _, err := primitive.CopyInt32SliceOut(t, addr, fds); err != nil {
        for _, fd := range fds {
            if file, _ := t.FDTable().Remove(t, fd); file != nil {
                file.DecRef(t)
            }
        }
        return 0, err
    }
    return 0, nil
}
```

Code Ex. 4

Netstack communication between the Sentry and the host

The use of a virtualized networking device in the Sentry to communicate with the host if a system call is needed adds to the levels of defense that gVisor is attempting to provide. In addition, the user space networking stack allows for gVisor to implement networking capabilities with the reduced set of system calls. This is done with use of only three additional system calls. If, however, more performance is required for networking, the contained application may use passthrough mode, which allows for use of the host's network stack implementation. Why not always use this? More performance is better, right? There is a tradeoff! Surprise, surprise, the increased performance of passthrough necessitates more whitelisted system calls, including increased file I/O like creating file descriptors, greatly increasing the attack surface of gVisor. This is due to the need for more system calls when using the host's networking stack to utilize all of the host's API. So, the general trend continues for less performance = more security.

Source - netstack



Connect in repo

And here is an example of the connect Linux syscall in userlevel netstack implementation

```
// gvisor/pkg/sentry/socket/netstack/netstack.go#L545

// Connect implements the linux syscall connect(2) for sockets backed by
// tpcip.Endpoint.
func (s *socketOpsCommon) Connect(t *kernel.Task, sockaddr []byte, blocking bool)
*syserr.Error {
    addr, family, err := socket.AddressAndFamily(sockaddr)
    if err != nil {
        return err
    }

    if family == linux.AF_UNSPEC {
```

```
        err := s.Endpoint.Disconnect()
        if _, ok := err.(*tcpip.ErrNotSupported); ok {
            return syserr.ErrAddressFamilyNotSupported
        }
        return syserr.TranslateNetstackError(err)
    }

    if err := s.checkFamily(family, false /* exact */); err != nil {
        return err
    }
    addr = s.mapFamily(addr, family)

    // Always return right away in the non-blocking case.
    if !blocking {
        return syserr.TranslateNetstackError(s.Endpoint.Connect(addr))
    }

    // Register for notification when the endpoint becomes writable, then
    // initiate the connection.
    e, ch := waiter.NewChannelEntry(nil)
    s.EventRegister(&e, waiter.WritableEvents)
    defer s.EventUnregister(&e)

    switch err := s.Endpoint.Connect(addr); err.(type) {
    case *tcpip.ErrConnectStarted, *tcpip.ErrAlreadyConnecting:
    case *tcpip.ErrNoPortAvailable:
        if (s.family == unix.AF_INET || s.family == unix.AF_INET6) && s.skType ==
linux.SOCK_STREAM {
            // TCP unlike UDP returns EADDRNOTAVAIL when it can't
            // find an available local ephemeral port.
            return syserr.ErrAddressNotAvailable
        }
        return syserr.TranslateNetstackError(err)
    default:
        return syserr.TranslateNetstackError(err)
    }

    // It's pending, so we have to wait for a notification, and fetch the
    // result once the wait completes.
    if err := t.Block(ch); err != nil {
        return syserr.FromError(err)
    }

    // Call Connect() again after blocking to find connect's result.
    return syserr.TranslateNetstackError(s.Endpoint.Connect(addr))
}
```

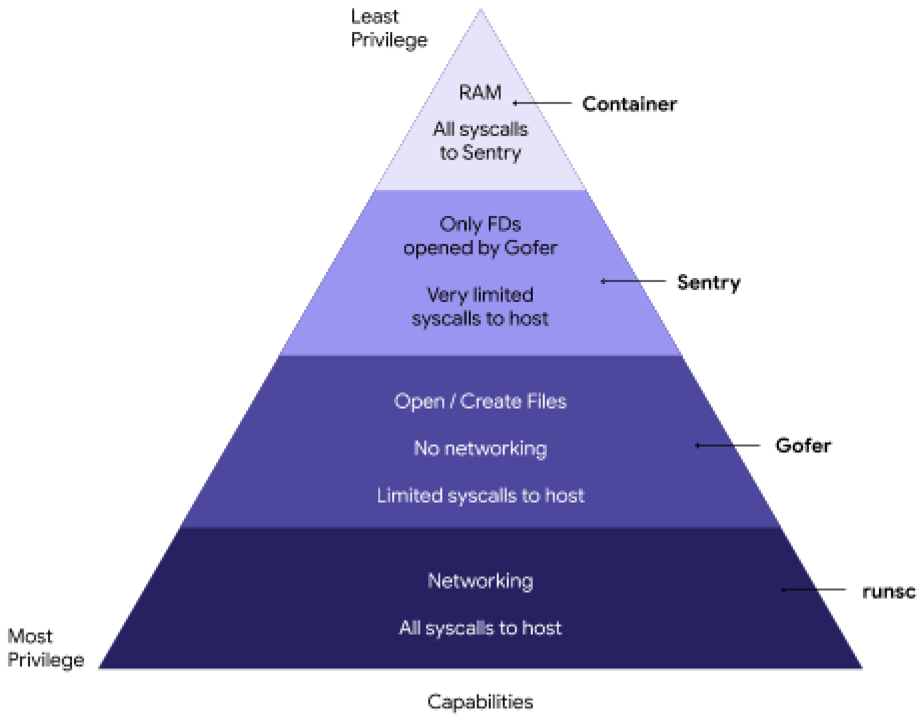
Security Fig. 2

Gofer

Another layer of the security model is the separation between the Sentry and file operations. The Sentry is able to have Gofer service its requests for file operations via 9P. The Sentry is not allowed to directly interact with the filesystem, and is in a constrained namespace. But what if it needs a file? The Sentry can communicate with Gofer via a socket with the 9P protocol and Gofer handles the file requests. In the interest of not adding more length to this novella, I will link here for a more in-depth Gofer explanation.

Security Conclusion

gVisor is able to acheive the level of security it has due to its many layers of defense, secure-by-default principle, and persistant assumption that the contained application is untrusted. The layers, consisting of limited system calls, restriction of file resource access, individual system call implementations, and the intercepting of application system calls, all contribute to the security-in-depth principle. They provide insurance for if the application is able to bypass one measure, another will be there to slow attacks down, or minimize the available attack surface in general. The secure-by-default principle can be seen in the choice to implement and use the user-level networking stack in the Sentry, opposed to the host's in passthrough mode, by default. Lastly, having all modules use the least priviledge possible ensures separation, as well as modules not having access to functionality that would increase the attack surface or vunerabilities, i.e. the container not being able to directly open fd's. If one module needs to perform a task that they are not able to perform, then there is a more trusted module to service the request.



Performance / Optimizations

In looking at the performance of gVisor, it is important to look at five main benchmarks:

1. Container startup/tear down
2. System call throughput
3. Memory allocation
4. File system access
5. Networking

An important point that must be discussed before comparing performance is which platform is used to handle system calls. The first option is *Kernel Virtual Mode (KVM)* which allows Linux itself to act as hypervisor by providing a loadable kernel module. The other option is *ptrace* which allows a process to intercept system calls being called. It should be noted that ptrace suffers from the highest structural costs by far.

The True Cost of Contanerization - Ethan G. Young, et al.

A paper written by Ethan G. Young, et al. at the Univeristy of Wisconsin ran experiments running `runc` against `runsc` in both ptrace and KVM mode. Their study examined the two systems in the five different benchmarks discussed previously.

Container Startup/Tear Down

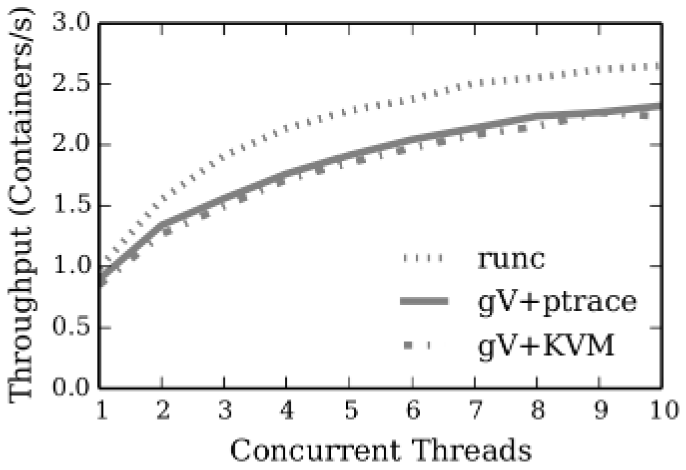


Figure 2: **Container Init/Teardown Performance.** *Varying numbers of threads (x-axis) create and destroy containers concurrently. The total system throughput is reported as containers-per-second on the y-axis.*

The results suggest that the differences between running gVisor with ptrace versus KVM for container initialization is neglible. In the context of runc however, there is about a 13% decrease in performance between runc and runsc. Although Google claims that gVisor is designed for use in machines with many containers, these results suggest that runc still has a slight edge.

System Call Throughput

In order to test the system call performace of gVisor, three versions of the same `gettimeofday` syscall was implemented: one invoking just the Sentry, one invoking the host OS, and the third invoking the use of Gofer. It is also important to note that gVisor was tested in both ptrace and KVM mode.

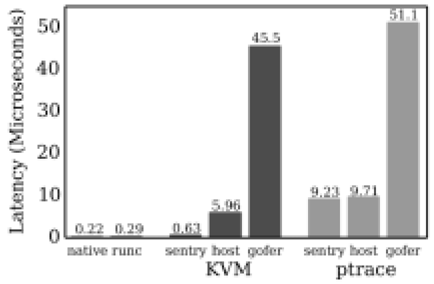


Figure 3: **System Call Overhead.** *The bars show the average latency for* `gettimeofday` *across 100M executions.*

The implication here is that even in gVisor's best case (running in KVM and only calling Sentry), the performance is still 2.8x slower. It is also clear from the results that calling to Gofer suffers from the worst performance. Compared to runc, Gofer runs between 156x and 175x slower depending on the platform.

Memory Allocation

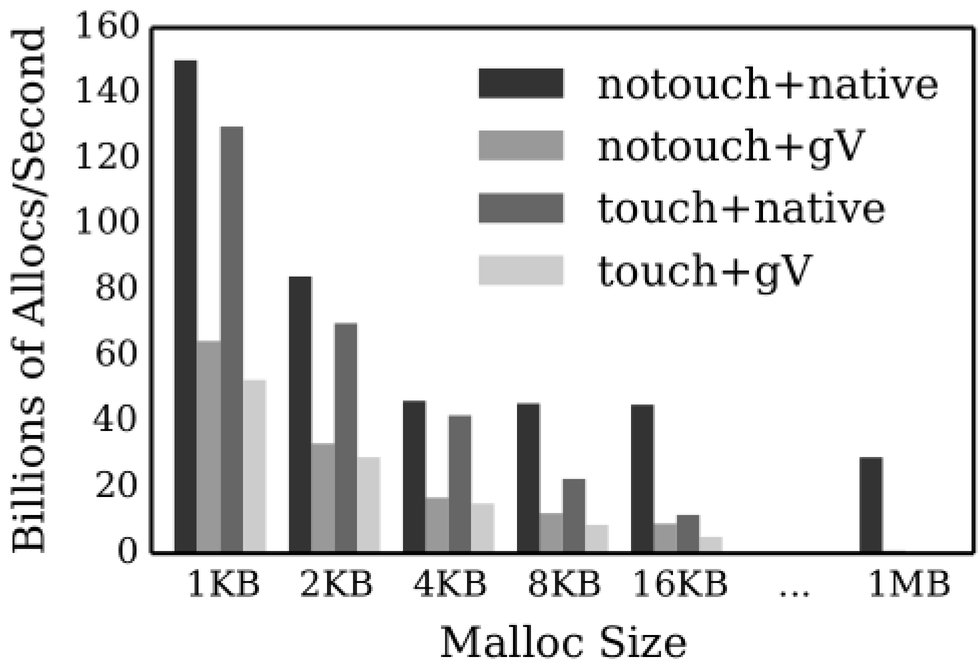


Figure 4: **Malloc Performance.** *Results are shown for both native and gVisor experiments, with and without touching the allocated memory.*

The key takeaway from these results is that gVisor achieves just 40% the allocation rate of native systems.

File System Access

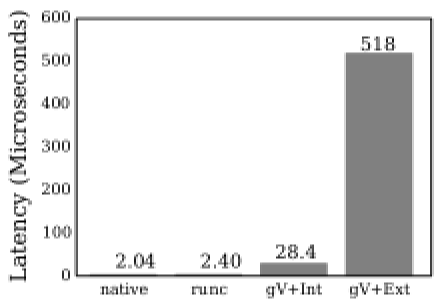


Figure 6: **Open and Close Latency.** *Results are an average over 100K consecutive accesses to the same file on native, runc, internal tmpfs (gV+Int), and external tmpfs (gV+Ext).*

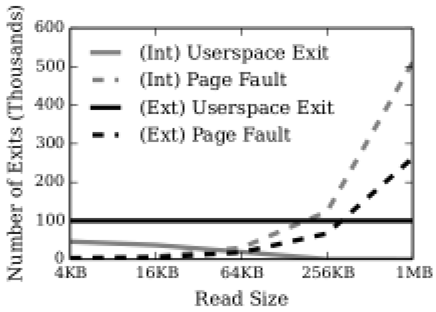


Figure 8: **KVM Exits for Read Workload.** *The x-axis shows the read size. The solid line represents the number of user-space exits performed by the KVM. The dotted line represents the number of exits by the KVM due to page faults.*

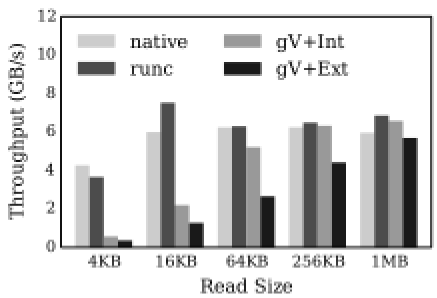


Figure 7: **Read Throughput (tmpfs).** *The x-axis shows the read size. The right two bars in each group are for gVisor.*

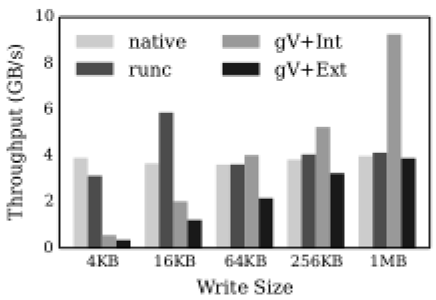


Figure 9: **Write Throughput (tmpfs).** *The x-axis shows the write size.*

As previously discussed, Gofer suffers from the largest performance tradeoff. This is even more evident in the results of using Gofer to access files. Opening and closing files on Gofer's external tmpfs is 216x slower than native compared to just 12x slower for access to Sentry's internal tmpfs.

Networking

gVisor uses its own network stack in order to safely and securely handle all networking. This is one area in particular that Google claims "is improving quickly". To test networking throughput, `wget` was called for file sizes of various sizes.

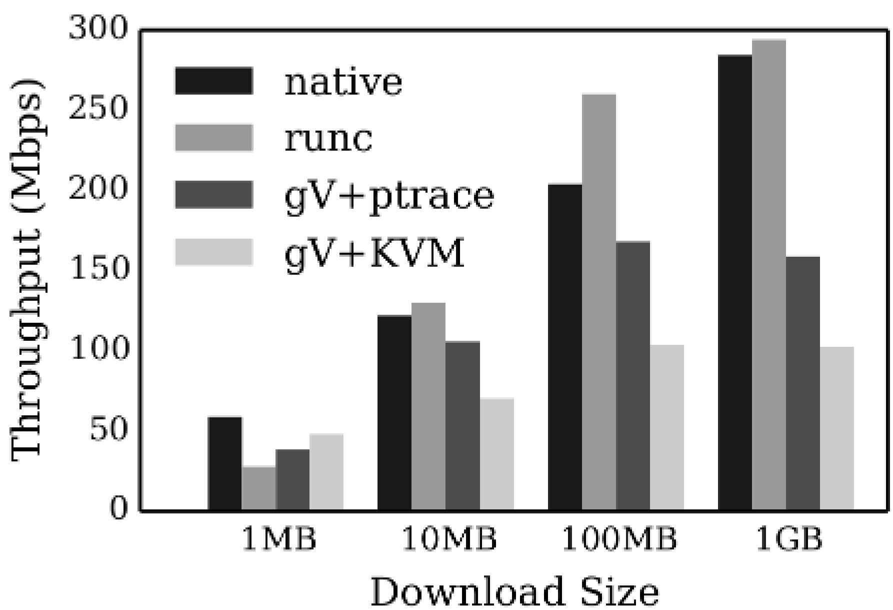


Figure 5: **Network Throughput.** *Throughput is measured by downloading files of varying sizes (x-axis) with* `wget`.

The results show that gVisor may handle small downloads well, relative to native performance, but as file sizes increase, gVisor fails to scale well.

Threading model

Goroutines are lightweight "green threads" managed by the Go runtime. These are used for individual Sentry tasks and the Gofer server model. See this paper comparing Linux Threads to Green Threads. This paper explains green threads in the Java runtime but the concept applies here as well for the Go runtime. Green threads are a userspace thread implementation that don't need a backing kernel thread. In this model, green threads are mapped to a single task and can be managed with little overhead. This is in contrast to the system calls required with linux threads. Thread activation also takes less overhead because Linux threads must create a corresponding execution entity within the kernel. When latency is important for an application, it is recommended for threads to be created at initialization rather than on demand. This explains the use of goroutines rather than native Linux threads throughout gVisor. Sources: Goroutines, resource model documentation

Gofer Server Synchronization

The below code `Lock()` function is the function used (as discussed above in the Gofer module) line 519 of gvisor/pkg/p9/server.go.

```
// line 72 /usr/local/go/src/sync/mutex.go
// If the lock is already in use, the calling goroutine
// blocks until the mutex is available.
func (m *Mutex) Lock() {
    // Fast path: grab unlocked mutex.
    if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked) {
        if race.Enabled {
            race.Acquire(unsafe.Pointer(m))
        }
        return
    }
    // Slow path (outlined so that the fast path can be inlined)
    m.lockSlow()
}
```

Source: go/src/sync/mutex.go

You can see an interesting Go optimization here where the fast path assumes the mutex is free and inlines that logic in the function. If the mutex is taken, `lockSlow()` is implemented in a different function because the blocking mechanism takes more logic and is therefore slow pathed.

Security-Performance Trade-offs of Kubernetes Container Runtimes - Viktorsson, et al.

Another study attempted to measure the performace of gVisor in more real world applications. All tests run in this study used the ptrace platform. Three different application's total throughput was tested:

1. **TeaStore:**

- TeaStore is a microservice benchmark that emulates a webstore and provides features that include browising, selecting, and ordering tea. The throughput is measured as the average requests per second throughput for eight available API operations.

2. **Redis:**

- Redis is an in memory data-store featuring data structures such as hashes, lists, sets, and more. The throughput is measure using requests per second of the `O(1) GET` operation.

3. **Spark:**

- Spark is a distributed general purpose computing framework for big data processing. The throughput is measured as the average number of primes found per second when finding all prime numbers in the first million integers.

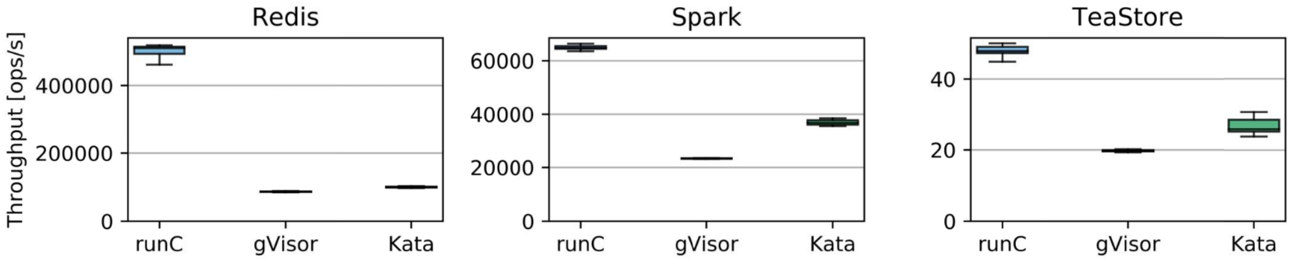


Fig. 3: Application performance – more is better – for each container runtime.

In testing TeaStore and Spark, gVisor has about 40-60% the throughput of runc. For Redis, it suffers dramtically at just 20% the throughput of runc. The poor performace in Redis is likely due to the fact that it is neither CPU nor memory demanding and thus its performance is based solely on the `GET` request to in-memory data. This suggests that Redis performance is largely based on networking throughput.

A Deeper Look into Memory Allocation in gVisor

gVisor's memory allocation system involves a two-level physical to virtual mapping where first Sentry requests memory chunks of 16MB increments from the host OS. Then, when an application running in the sandbox requests memory (using `mmap()`), Sentry allocates a portion of the 16MB chunk for the application.

From Host to Sentry

First we'll look at the sequence of code that allows Sentry to get memory from the host OS. `Allocate()` updates a `MemoryFile` which is a mapping of a chunk of memory from the host. This is the struct that Sentry will later use to find available pages of memory to allocate to an application.

```
// Line 381 of pgalloc.go
// Allocate returns a range of initially-zeroed pages of the given length with
// the given accounting kind and a single reference held by the caller. When
// the last reference on an allocated page is released, ownership of the page
// is returned to the MemoryFile, allowing it to be returned by a future call
// to Allocate.
//
// Preconditions: length must be page-aligned and non-zero.
func (f *MemoryFile) Allocate(length uint64, kind usage.MemoryKind)
```

```
(memmap.FileRange, error) {

    // ...

    // Align hugepage-and-larger allocations on hugepage boundaries to try
    // to take advantage of hugetmpfs.
    alignment := uint64(hostarch.PageSize)
    if length >= hostarch.HugePageSize {
        alignment = hostarch.HugePageSize
    }

    // Find a range in the underlying file.
    fr, ok := findAvailableRange(&f.usage, f.fileSize, length, alignment)
    if !ok {
        return memmap.FileRange{}, syserror.ENOMEM
    }

    // ...

    if f.opts.ManualZeroing {
        if err := f.manuallyZero(fr); err != nil {
            return memmap.FileRange{}, err
        }
    }
    // Mark selected pages as in use.
    if !f.usage.Add(fr, usageInfo{
        kind: kind,
        refs: 1,
    }) {
        panic(fmt.Sprintf("allocating %v: failed to insert into usage set:\n%v",
fr, &f.usage))
    }

    return fr, nil
}
```

From Sentry to Application

When an application running in gVisor calls `mmap()`, first the `Mmap()` syscall is invoked:

```
// Line 42 of sys_mmap.go
func Mmap(t *kernel.Task, args arch.SyscallArguments) (uintptr,
*kernel.SyscallControl, error)
```

It is important to note the two arguments to the function: `t *kernel.Task` and `args arch.SyscallArguments`. A `Task` represents an execution thread in an un-trusted app. This includes thread-specific state such as registers. `SyscallArguments` include the length of the memory region requested and a pointer to the memory. These arguments will later be stored in an `MMapOpts` object inside `MMap()`.

From here, `MMap()` is invoked in Sentry.

```
// Line 75 of syscalls.go
func (mm *MemoryManager) MMap(ctx context.Context, opts memmap.MMapOpts)
(hostarch.Addr, error) {
```

This function takes in `Context`, which represents the thread of execution, as well as the `MMapOpts` that was created in the previous function.

Inside of `MMap()`, `createVMALocked` is called on line 122 which is where a new VMA is allocated.

```
// Line 33 of vma.go
func (mm *MemoryManager) createVMALocked(ctx context.Context, opts
memmap.MMapOpts) (vmaIterator, hostarch.AddrRange, error) {

    // ...
    // Line 38

    // Find a usable range.
    addr, err := mm.findAvailableLocked(opts.Length, findAvailableOpts{
        Addr:     opts.Addr,
        Fixed:    opts.Fixed,
        Unmap:    opts.Unmap,
        Map32Bit: opts.Map32Bit,
    })

    // ...
    // Line 55

    // Check against RLIMIT_AS.
    newUsageAS := mm.usageAS + opts.Length
    if opts.Unmap {
        newUsageAS -= uint64(mm.vmas.SpanRange(ar))
    }
    if limitAS := limits.FromContext(ctx).Get(limits.AS).Cur; newUsageAS > limitAS
{
        return vmaIterator{}, hostarch.AddrRange{}, syserror.ENOMEM
    }

    if opts.MLockMode != memmap.MLockNone {
        // Check against RLIMIT_MEMLOCK.
        if creds := auth.CredentialsFromContext(ctx);
!creds.HasCapabilityIn(linux.CAP_IPC_LOCK, creds.UserNamespace.Root()) {
            mlockLimit := limits.FromContext(ctx).Get(limits.MemoryLocked).Cur
            if mlockLimit == 0 {
                return vmaIterator{}, hostarch.AddrRange{}, syserror.EPERM
            }
            newLockedAS := mm.lockedAS + opts.Length
            if opts.Unmap {
                newLockedAS -= mm.mlockedBytesRangeLocked(ar)
            }
            if newLockedAS > mlockLimit {
```

```
                return vmaIterator{}, hostarch.AddrRange{}, syserror.EAGAIN
            }
        }
    }

    // ...
}
```

`createVMALocked()` finds a mappable region of memory to allocate. One of the important checks is to see if the `Context` has access to this region and can allocate more memory.

Tradeoffs of gVisor's Memory Allocation

Due to the two level page table system, applications requesting small pieces of memory (relative to the size requested from the host by Sentry) suffer in performace. As the size of the memory requested by an application increases, the performance increases because less work in Sentry must be done to further divide the 16MB chunk from the host OS. However, it should be noted that memory allocation of any size is not very fast in gvisor relative to native Linux containers.

Blending Containers and Virtual Machines: A Study of Firecracker and gVisor - Anjali, et al.

This next paper studied memory performance differences between native Linux with no isolation, Linux containers, and gVisor (using KVM-mode).

Because of gVisor's two level page tables, Sentry requests memory from the host in 16MB chunks in order to reduce the number of `mmap()` calls to the host. When 1GB of memory is requested by the host application, there will be exactly 64 `mmap()` calls to the host.

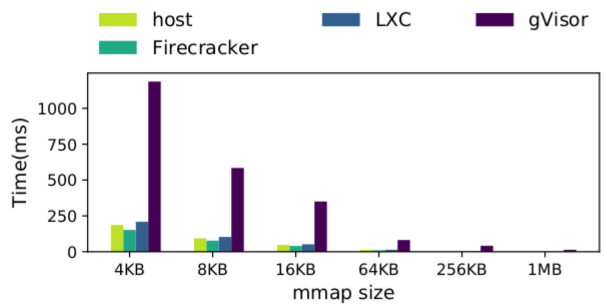


Figure 14. Total allocation time (without munmap) for 1GB

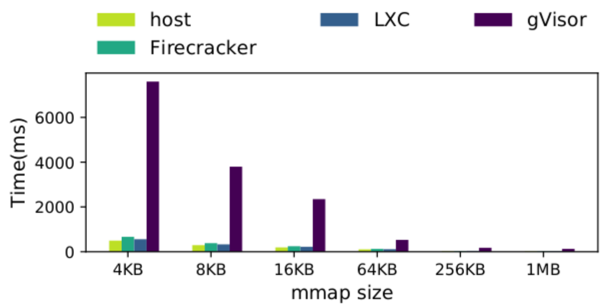


Figure 16. Total allocation+unmap time for 1GB

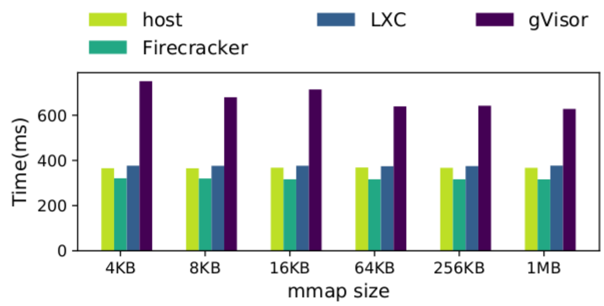


Figure 15. Total touch time (without munmap) for 1GB

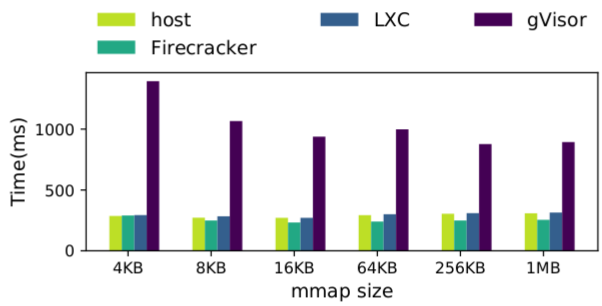


Figure 17. Total touch time (with munmap) for 1GB

The test performed called `mmap()` with varying sizes ranging from 4KB to 1MB for a total of 1GB of memory. The results show that when allocating 4KB pieces gVisor performs about 16x slower than host Linux and Linux containers. When allocating 64KB chunks, the gap lessens by almost half and gVisor is only 8-10x slower. In the case of comparing to gVisor, the difference between host Linux and Linux containers is negligble.

This is an important implication as there is a trend between gVisor's memory allocation performace and the size of the request: as size increases, gVisor's gap to Linux grows smaller. This is likely due to the two-level page table system implemented in gVisor. As an application's memory request grows closer to 16MB, less work in Sentry is being performed to further split that chunk into smaller pieces for the applications running in the sandbox.

Performace Conclusion

From the studies presented, it is clear that if performance is a concern, gVisor is not a good fit for applications requiring heavy use of syscalls, heavy use of memory, heavy use of networking, nor heavy use of file system accesses. gVisor instead is a good fit for lightweight, serveless applications. Because gVisor does not suffer a significant performance loss in building, deploying, and destroying containers compared to standard Linux containers, secure containers can be quickly created for lightweight applications as needed.

Subjective Opinions

- Sam
  - gVisor is a really cool solution to a specific problem. It's great at its stated goal, providing a secure sandbox for testing untrusted code. In doing so, there are some substantial tradeoffs. Perfomance is slow, but in this case, I think that's ok given the level of security provided. I found the use of the separately sandboxed Gofer for file access to be a very interesting way to protect host files. I am most curious if communication with Gofer could be further optimized to improve performance.
- Jake
  - While gVisor certainly doesn't have the greatest performance for many usecases, I think its a good first step towards securing cloud infrastructure. Maybe right now it can only be feasible for serverless frameworks (as designed), it could eventually evolve or motivate a higher performing implementation that could work for larger and more diverse workloads. Even as is, it could be extremely useful for any application if you could just offload the most critically sensitive logic to gVisor. I think security is often traded for performance but it's great to see a project with security as the number one goal. Given time and effort, gVisor and similar projects can get faster. I feel like this is a better route to take than trying to constantly patch new security vulnerabilities in a fast system. I also really like that the language is written in Go rather than C because many existing Linux vulnerabilities come about from problematic C code and I'm excited to see the relatively young Golang evolve further.
- Jack
  - gVisor's lack of performance in many areas is certainily a large pill to swallow on first look comparing it to native Linux Containers. However, if security is of utmost concern, then the many tradeofs made in gVisor soon become worth it. The two-level memory allocation system is a clever way of reducing the number of calls needed to be made down to the host OS. Allowing Sentry to request memory in chunks allows gVisor's footprint to remain small when low amounts of memory are required by an application. It would be interesting to see how the size of the chunks requested by Sentry affects overall performance. Would dynamically increasing or decreasing the 16MB chunk based on the expected use of the container have a significant effect on performance?

- Will
  - gVisor is a very secure container implementation with many layers of defense in depth, and it shows (preventing security vulnerabilities that docker, for example, was subsceptible to). However, there is a consequential and significant performance hit. I don't necessarily think this is a nail in the coffin, though. To acheive the level of security that gVisor was intending to reach, there are necessary tradeoffs that had to be made - like tracing all system calls and redirecting to be intercepted. That simply cannot be cheap no matter which way you look at it. But that is not the point of gVisor. The core component is security, and it does it very well.
- Jon
  - The performance hits gVisor takes as an effect of the features of its security oriented design seem so significant that gVisor would be entirely useless in any setting other than a serverless framework operating on the principle of functions as a service. This is exactly what it was designed for, so this isn’t a huge insult, but it certainly is not a very versatile system and modifying it to be of use in other environments does not seem like a feasible task to undertake. The security principles of the system are exceptional and it is clear how gVisor would be far better for fully isolating applications from each other than traditional container sandboxing. For its intended purpose gVisor provides much needed security and isolation, and the performance tradeoffs with security are worth it. When used in the type of systems it was intended for, the improved construction and deconstruction time of containers should have a larger impact on performance than the issues with performance gVisor faces at runtime - in all, gVisor adequately meets its goals and successfully tackles the problems it tries to solve with container sandboxing with acceptable tradeoffs, making it a very good system for serverless framework cloud computing.