Alto: Lightweight VMs using Virtualization-aware Managed Runtimes

James Larisch, James Mickens, Eddie Kohler
Harvard University
jameslarisch,mickens@g.harvard.edu; kohler@seas.harvard.edu

ABSTRACT

Virtualization enables datacenter operators to safely run computations that belong to untrusted tenants. An ideal virtual machine has three properties: a small memory footprint; strong isolation from other VMs and the host OS; and the ability to maintain in-memory state across client requests. Unfortunately, modern virtualization technologies cannot provide all three properties at once. In this paper, we explain why, and propose a new virtualization approach, called Alto, that virtualizes at the layer of *a managed runtime interface.* Through careful design of (1) the application-facing managed interface and (2) the internal runtime architecture, Alto provides VMs that are small, secure, and stateful. Conveniently, Alto also simplifies VM operations like suspension, migration, and resumption. We provide several details about the proposed design, and discuss the remaining challenges that must be solved to fully realize the Alto vision.

CCS CONCEPTS

• **Security and privacy → Virtualization and security**; • **Computer systems organization → Cloud computing**;

KEYWORDS

Virtualization, memory management, minimal TCBs, datacenters

ACM Reference Format:
James Larisch, James Mickens, Eddie Kohler. 2018. Alto: Lightweight VMs using Virtualization-aware Managed Runtimes. In *15th International Conference on Managed Languages & Runtimes (ManLang'18), September 12–14, 2018, Linz, Austria.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3237009.3237022

1 INTRODUCTION

In cloud computing, tenants decompose their applications into individual components like web servers, databases, and publish/subscribe queues. The datacenter operator provides a virtual machine to each component, exposing a portion of the datacenter's underlying physical hardware to the VM. Ideally, datacenter VMs should be *stateful, lightweight, and strongly isolated.* Stateful computations allow applications to maintain in-memory data across multiple requests, without requiring the data to be recreated for each request, or pulled from the storage layer. Lightweight VMs allow a datacenter operator to quickly and efficiently launch new VMs, migrate old VMs across physical machines, and suspend VMs to disk for later resurrection. Strongly-isolated computations give confidence to both datacenter operators and datacenter tenants that misbehaving VMs can only damage themselves.

Unfortunately, no model for datacenter virtualization satisfies all three properties.

- Hardware-level virtualization (e.g, Xen [8]) provides strong isolation and support for persistent in-memory state. However, VM sizes are large, making VM operations like snapshotting and migration expensive.
- OS-level containers (e.g., Docker [16]) substantially reduce VM sizes while keeping support for persistent RAM state. However, in exchange for lightweight, stateful computations, isolation becomes worse—container partitions are enforced by a constellation of security checks dispersed throughout a complex kernel that is shared by multiple containers. A related side effect of poor isolation is that container snapshotting and resurrection involves fragile inspection of process state that was not designed for efficient enumeration and serialization.
- Stateless functions like Amazon Lambda [3] dramatically reduce the developer effort needed to scale an application. However, in exchange for lightweight computations, these approaches sacrifice isolation; functions execute atop containers, and thus inherit the security problems associated with OS-level virtualization. Lambda-style approaches also sacrifice statefulness, which is required by many services (in particular, those that have a notion of session state).

In this paper, we suggest a new approach: *datacenters should virtualize at the level of a carefully-designed managed runtime interface.* This interface, called Alto, is intentionally designed to provide stateful, lightweight, and strongly-isolated computations. All of the code inside an Alto VM is written in a managed language which targets the Alto managed API. Using memory allocation and garbage collection which prioritizes contiguous allocation, Alto permits an entire VM to be snapshotted or restored in a single `memcpy()`.

Of course, a managed process may consist of more than just its private RAM state—the process may reference external, possibly-shared resources like the file system, or memory pages shared by multiple processes. To simplify a process's ability to detach from, and reattach to, those resources, Alto's managed runtime internally represents external resources as network-connected Alto VMs; these VMs can themselves be suspended, resumed, and migrated.

Alto computations are stateful because they are expressed using managed code whose RAM data is long-lived. Alto computations are secure because the Alto managed interface is smaller than an OS-level interface, and has simpler semantics. Alto computations

ManLang'18, September 12–14, 2018, Linz, Austria
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6424-9/18/09.
https://doi.org/10.1145/3237009.3237022

are lightweight because memory images are small, and because single-`memcpy()` state capture makes snapshotting, migration, and resumption efficient. The Alto project is in its early stage, but we believe that a single physical machine can simultaneously run an order of magnitude more Alto VMs than Xen VMs or Docker containers.

We also believe that Alto will enable fundamentally new kinds of applications. For example, Alto makes it easier for companies to scalably provide *per-user VMs*, even for very popular services that have many users. Strongly-isolated, per-user VMs allow for easier compliance with regulations (or user demands) for control of datacenter-side state at the granularity of individual users [18].

2 BACKGROUND

In this section, we describe the advantages and disadvantages of preexisting approaches for virtualization.

2.1 Heavyweight VMs

Traditionally, the virtualization layer has been defined at the hardware level [1, 8, 34]. A virtual machine contains the code and data for the application of interest; the virtual machine also contains the code and data for the guest OS which directly interacts with the application. A hypervisor runs directly atop the physical hardware, mediating how the guest OS interacts with the physical hardware. Since the hypervisor interposes on sensitive instructions that manipulate page tables and IO devices, the hypervisor can safely partition a single set of physical resources into multiple sets of virtual resources.

Traditional VMs provide strong isolation, and allow arbitrary applications to run atop arbitrary guest OSes. Unfortunately, VM snapshots are large (often hundreds of MBs), because those snapshots must contain the RAM state for *all* processes in the virtual machine, including the operating system; ideally, a VM snapshot would contain precisely the set of RAM pages that belong to an application of interest. Because traditional hypervisors virtualize at the hardware layer, snapshots must also capture low-level device state which can be tricky to properly serialize and restore [17, 25, 39, 43]. Resurrecting a VM is slow, often taking multiple seconds, because snapshots are large and require non-trivial time to read into memory. In the context of VM migration, the synchronous overhead of snapshot generation, transmission, and resumption can be partially masked using a variety of techniques, e.g., by migrating snapshot state in the background, as the source VM continues to operate [11, 14, 37]. Unfortunately, the large sizes of VM images, and the intricacies of multiplexing physical hardware across multiple VMs, still restrict the number of VMs that a physical host can run. For example, a VMware ESXi hypervisor has a limit of 1024 VMs [46].

2.2 Lightweight Containers

Container systems virtualize at the OS level instead of the hardware level. For example, a process in a Docker container [16] interacts with the outside world using the standard Linux system call interface; however, the Linux kernel internally uses namespaces [26] and cgroups [9] to restrict the process's view of sensitive resources like PIDs, the file system, and network interfaces. All processes in a container are given the same restricted view.

Since a Docker container is essentially just a set of Linux processes, Docker implements container snapshotting via CRIU [13], a standalone tool for capturing the state of a Linux process. CRIU uses the `/proc` file system and the `ptrace()` system call to gather the necessary checkpoint state for a set of processes. At restore time, CRIU uses `fork()` to recreate the appropriate process trees. Then, CRIU maps the necessary code and heap pages into the newly-created address spaces. Finally, CRIU performs a variety of fix-ups to re-`bind()` sockets, `chdir()` threads into the appropriate directories, reset timers, and so on.

Containers have much smaller snapshots than VMs; unlike a VM snapshot, a container snapshot does not include state for a guest OS. However, container frameworks have two security problems:

- The lack of a per-container guest OS means that containers which run on a single physical host share the same host OS. This arrangement leads to fate-sharing—if one container manages to corrupt the host OS, then all containers are at risk [27, 44].
- An OS interface like POSIX is wide and complex. To implement container-based isolation, security checks must be sprinkled across a large number of code paths in the kernel. The result is that the container "security monitor" is decentralized, and must defend a large threat surface.

Containers have low portability, since a container which targets (say) the Linux interface either cannot run on a Windows machine, or must execute atop complex, slow middleware for system call emulation [2]. Container operations like snapshotting and resumption are faster than the equivalent VM operations, but are still tricky to get right, since OS-level process abstractions were not designed with virtualization in mind. So, tools like CRIU must engage in a variety of low-level skulduggery (e.g., involving `ptrace()`) to identify container state, properly snapshot it, and restore it later. This complexity hurts the efficiency of snapshot and resumption.

2.3 Stateless Functions

Web services have traditionally used long-lived datacenter computations that run inside of VMs or containers. In contrast, Amazon Lambda [3] structures server-side code as a group of short-running functions. Each function is triggered by an event (e.g., the reception of an HTTP request, or the mutation of a storage value in S3). Lambda functions maintain no RAM state between invocations, so functions can only modify application state by writing to persistent network-attached storage. Each function is executed within a container that the datacenter may or may not discard between invocations of the function.[1] Besides Lambda, other examples of stateless, event-driven architectures include OpenLambda [21], OpenWhisk [4], and Google Cloud Functions [20].

For simpler applications, or those with less stringent performance requirements, a stateless architecture has a variety of benefits. From the perspective of application developers, writing code is simpler. For example, with persistent storage as the first and only repository for canonical application state, many of the consistency problems incurred by traditional VM-based designs are

[1]In addition to writing to network-attached local storage, a function can write to the local `/tmp` directory. However, the contents of the directory will be lost if the function's old container is destroyed between function invocations [47].

obviated—there is no need to synchronize the RAM state of multiple application processes. A stateless architecture also makes scaling easier; since functions are cheap to launch and short-running, the datacenter operator can automatically launch the requisite number on-demand, without requiring developers to reason about how to provision heavyweight VMs. Since stateless functions require less RAM and less CPU than long-lived VMs, datacenter operators can sell stateless functions at lower prices than VMs.

Despite these advantages, stateless functions are a poor fit for many kinds of applications. For example:

- Stateless functions are unable to cache database query results across function invocations. Thus, server-side performance will suffer if client requests exhibit locality.
- If applications are session-oriented, then the lack of long-lived RAM state hurts performance. For example, consider a smartphone app for ride sharing. Without the ability to bind a particular client's requests to a long-lived server computation, each request will invoke a separate ephemeral function; that function must repeatedly fetch session state from persistent storage, rebuild the appropriate in-memory data structures for the session, and then write the updated state to persistent storage when the function terminates.
- The start-up latencies for a stateless function can be high if the datacenter has discarded the last container that the function used [47, 49]. Unpredictable start-up times make it hard for application developers to predict user-perceived application latencies. In contrast, long-lived VMs or containers can persist important data in RAM, offering faster response times.

Datacenters use containers to execute functions [21, 47]. As a result, platforms like Lambda inherit the security issues that are shared by all container-based systems.

2.4 Unikernels

A unikernel [31–33] is a single-purpose executable that is optimized for small memory footprints and high performance. The functionality of a traditional OS is split into a set of libraries; application code is compiled, and then linked with the appropriate subset of libraries, to create the final unikernel. The resulting VM has a single address space, and no distinction between privileged and unprivileged code, although language-level isolation can be provided if all code is written in a strongly-typed language like OCaml [32].

Unikernels explicitly eschew backwards compatibility with legacy code and traditional programming environments like POSIX. Many unikernels also target limited types of applications; for example, ClickOS [33] only targets code that runs on middleboxes, with the only supported IO being to the network. A unikernel application that requires more functionality also requires more libraries, leading to a large OS footprint and recapitulating the problems of other virtualization schemes. This makes unikernels a good fit for low-level applications, such as packet forwarding, but less good for applications that use a wide range of conventional OS functionality, such as many of our target applications. Alto's goal is to support general-purpose computation, with traditional process isolation via address spaces. Alto willingly trades some computational performance to achieve this goal.

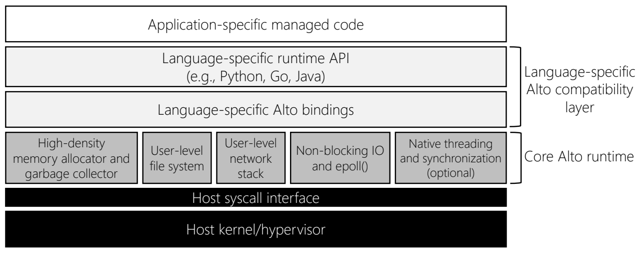

Application-specific managed code
Language-specific runtime API (e.g., Python, Go, Java)
Language-specific Alto bindings
Language-specific Alto compatibility layer
High-density memory allocator and garbage collector
User-level file system
User-level network stack
Non-blocking IO and epoll()
Native threading and synchronization (optional)
Core Alto runtime
Host syscall interface
Host kernel/hypervisor

Figure 1: Our preliminary design for Alto. Applications (i.e., virtual machines) are written using standard managed languages, but Alto's custom runtime ensures that VMs are small, and that VM snapshotting, migration, and resumption are fast.

3 ALTO: DESIGN CHALLENGES

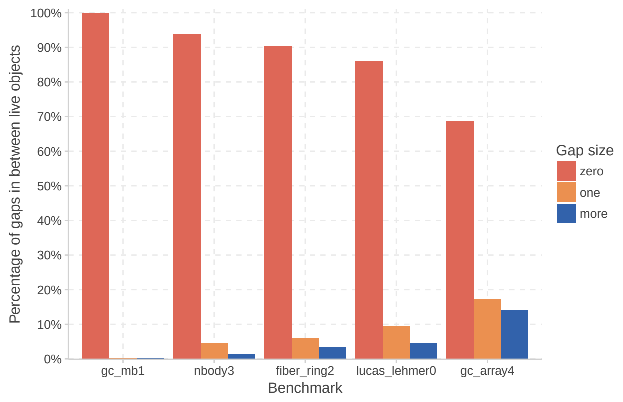
Alto's goal is to provide VMs that are small, efficient to manage, and yet powerful enough to support complex applications. To realize this vision, Alto must solve a variety of design challenges. In this section, we describe some of these challenges, and propose some initial solutions. Figure 1 provides a high-level overview of the Alto design that we discuss below.

Challenge 1: What managed runtime features are most important to enable compact VMs? The most obvious desiderata are a memory allocator and a garbage collector that prioritize *high-density liveness*. High-density liveness means that, at any given moment, the active parts of VM memory (e.g., the memory belonging to live stack frames and heap objects) are located in a contiguous memory region that has minimal internal fragmentation. High-density liveness enables single-`memcpy()` snapshotting and resurrection of application-visible managed state.

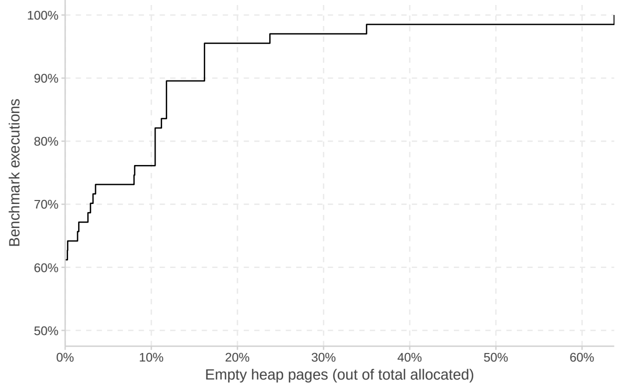
There is a tension between the density of live objects, and the runtime overhead which is needed to achieve that density. For example, high density is trivially achieved by running the garbage collector frequently, and aggressively compacting the live objects. Unfortunately, garbage collection is pure overhead from the perspective of application-level code. This naïve approach is consequently unattractive, even if long-lived objects are checked for liveness less frequently, as in a generational garbage collector.

Figure 2 shows that if a traditional managed runtime does not aggressively compact, high-density memory liveness will not occur. If compaction is deferred until immediately before a snapshot is taken, as done by some Smalltalk runtimes [41], then *on-disk* snapshots will be small; however, Alto also requires high-density *in-memory* heaps.

Proposed solution: Site-specific memory management. Alto will use *trace-based custom allocators for hot paths* to handle memory management. Much like traditional just-in-time compilers emit native code for a frequently-executed computational path, a trace-based custom allocator would use dynamic memory profiling to guarantee high-density liveness for the objects allocated by a particular hot path. Prior work has already shown that dynamic profiling of objects created by a particular allocation site can guide the decision to have that site immediately allocate objects from the



(a) Ruby's allocator defines a "page" as a 16 KB block of contiguous memory. Inside a page, Ruby allocates a 40 byte slot to each Ruby-level object, representing the object as a tagged union. Ruby's garbage collector does not perform compaction, so inside a page, some slots may lie unused at any given time. These bytes represent wasted space if copied to storage during a snapshot operation, or transferred over the network during a VM migration; in the extreme, a page might only contain dead objects. The graph above shows that, within a page that has at least one live object, slot utilization is often high, but not always. The graph depicts results for five representative benchmarks, each of which was run five times. For example, pages in the `gc_mb1` benchmark enjoy 100% slot utilization; in contrast, a `gc_array4` page has a 32% chance that the gap between two live objects is at least one slot.



(b) We ran each of the 65 benchmarks five times; at the end of each trial, we measured the fraction of pages with no live objects. The graph above shows a CDF of the fraction across all trials. Although 75% of trials ended with fewer than 10% dead pages, a tenth of the executions left at least 17% dead pages, and 5% of executions left at least 24% dead pages. This kind of long-tail behavior is undesirable. Modern web services often handle a single high-level request by spawning tens or hundreds of subtasks, with the overall response time determined by the slowest individual subtask [15, 23, 48]; ideally, Alto could avoid a long-tail in dead memory pages to avoid long-tail latencies for VM management operations.

Figure 2: Memory density in Ruby v2.5.1 (which uses a non-compacting garbage collector). Workloads came from the Ruby Benchmark Suite [10]. All experiments in this paper ran on an 2.8 GHz eight-core machine with 16 GB of RAM.

old generation memory pool [12]. We believe that, using similar instrumentation, the Alto runtime can optimize memory allocation patterns *across all of the sites in a hot path*, ensuring high-density liveness when the path has finished execution.

Proposed solution: Mingle stack pages and heap pages. Alto can also benefit from a unified allocator for heap objects and thread stacks; by interleaving stack frames with heap objects, density will improve. Inside an Alto VM, each thread will use a segmented stack which grows and shrinks on demand, e.g., in 2 KB increments. Thus, each function call is a potential allocation site, and each function return is a potential deallocation site.

Segmented stacks enable a single process to contain many threads. This property is useful for a managed runtime that targets server-side code. However, Go's experience with segmented stacks [35] indicates that the approach requires careful design—without runtime tuning of allocation decisions, a thread may repeatedly increase and then decrease its stack size, e.g., in response to a function being called in a loop. Using trace-based custom allocators, we hope to avoid such problems. For example, to handle T threads which exhibit the top-of-stack "flapping" mentioned above, Alto can allocate the T top-of-stack areas from a pool of $P < T$ regions; by empirically observing the best value for P, Alto can maximize the likelihood that all P regions are live at any given moment.

Challenge 2: How should VM state be represented? At a high level, a process (managed or otherwise) has two kinds of state: tightly-bound state and loosely-bound state. Tightly-bound state corresponds to private state that is not shared with other processes. Examples of such private state include executable code pages, static data pages, heap pages, open file descriptors, and TCP connection state. Loosely-bound state may be shared by other processes, and is supervised with the help of the operating system. Examples of loosely-bound state include the file system, shared memory pages, pipes, and graphics buffers. A single Alto application will possess both tightly-bound and loosely-bound state. How should the Alto runtime represent this state?

Proposed solution: User-level management of traditionally in-kernel state. In a traditional Linux process, state involving IO management is scattered across kernel memory and user memory. For example,

- Cached file data may reside in the kernel-level buffer cache, as well as user-level buffers managed by libc. The kernel also stores file descriptor metadata like the current seek position.
- For each TCP connection, the kernel maintains send and receive queues. The kernel also tracks statistics related to congestion control. User-level code often maintains additional buffers to pre-process data to send, or post-process data that has been received.

To simplify VM operations like snapshotting, Alto will hoist as much kernel state as possible into the VM itself. For example, using DPDK [30], an Alto VM can implement a user-level networking stack, allowing packet processing to completely bypass the kernel [24, 29]. Alto can also leverage FUSE [19] to implement user-space file systems.

A key research challenge is ensuring that these user-level subsystems have high-density memory liveness. File systems often use slab allocators, and DPDK requires hugepage allocations. Both approaches can lead to internal fragmentation.

Proposed solution: Sharing state by sharing VMs. Alto will place shareable state inside of a VM, such that state sharing leverages Alto's preexisting mechanisms for cross-VM communication. In the most radical decomposition, the contents of a single traditional Unix process would be split across multiple VMs. For example, consider a simple echo server which receives a message over TCP, logs the message to a file, and then echoes the message over the TCP connection. Alto could define one VM for the application's code, and a separate VM for the application's heap data, and another VM to store the application's network state, and yet another VM for the file system. Each VM would communicate via a publish/subscribe mechanism, somewhat reminiscent of the communication between a microkernel's servers.

This approach has several advantages. Explicitly representing each piece of state as an addressable server makes state enumeration and state serialization easier. These benefits are particularly noticeable for multi-component applications like an N-tier web service. For example, migrating all or part of the distributed application can be accomplished by enumerating the individual Alto VMs in the application, and then migrating the appropriate ones. Aggressive decomposition into multiple VMs also facilitates the independent modification of a subset of VMs. For example, hot updating of code has traditionally been a complex endeavor [6, 36]; clean separation of a program's code, stack, and heap can make dynamic updates easier.

Even if developers colocate heap, stack, and code pages inside of a single VM, the fact that each type of content is explicitly nameable and enumerable will assist with VM suspension, snapshotting, and resurrection. Alto wants to avoid the complex, fragile incantations that CRIU must invoke to discover application state. In principle, host OSes could be modified to support a single system call that, when invoked, would read or write all of the information that CRIU currently manipulates via `ptrace()` and `/proc`. However, the hypothesized system call would over-approximate the amount of state in an Alto VM—only the Alto runtime would know (for example) which allocated heap pages are actually live. Furthermore, information at the granularity of `ptrace()` and `/proc` is insufficiently expressive to perform ensemble manipulations of a multi-VM program like an N-tier web server.

Challenge 3: How should Alto VMs be addressed? As described above, Alto allows the stack, heap, and code of a single Alto-level process to be colocated in the same VM, or distributed across several VMs. We refer to the Alto-level process as a *logical VM*; the one or more underlying VMs are the *concrete VMs*. How should logical VMs and concrete VMs be named and made discoverable?

Proposed solution: High-level Alto namespaces backed by IP addressing. Alto will assign an IP address to each logical VM. The associated concrete VMs will reside in an Alto-managed namespace, with each concrete VM exporting descriptive metadata using OpenAPI [38], a format for describing RESTful services. Ensembles of logical VMs (e.g., in a multi-tier web service) will communicate with each other using publish/subscribe channels [5, 40].

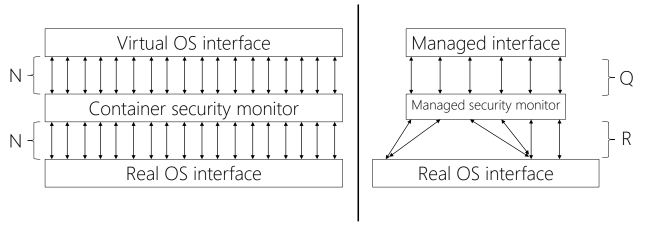


Figure 3: The left side represents a container technology like Docker. The right side represents Alto. Alto's threat surface should be smaller if $Q < N$, and/or $R < N$.

Challenge 4: How does Alto minimize the threat surface exposed by the managed interface, while still allowing rich applications to run atop that interface? Alto does not directly expose the wide, difficult-to-secure POSIX interface to application code. However, Alto's managed runtime must ultimately issue host-level system calls. Thus, there is a tension between the expressiveness of Alto's managed API, and the security of that interface. Figure 3 provides a visual intuition for the dilemma. In a standard container system, there is essentially a 1-1 mapping between the virtual OS interface exposed to containers, and the real OS interface which implements the virtual one. This 1-1 mapping implies that the threat surface for the container interface is the entire OS interface. In contrast, the Alto managed runtime can be selective about which host resources to expose; the runtime can also be judicious about defining higher-level abstractions for those resources. Intuitively, Alto's threat surface is smaller than that of the host OS if (1) the managed runtime only uses a subset of the full system call interface, and/or (2) Alto's higher-level interface is more amenable to security monitoring than the host's low-level system call interface. How can we guarantee conditions (1) and/or (2)?

Proposed solution: Decrease the number of host OS code paths that must be trusted. By leveraging user-level code to handle the bulk of IO processing, an Alto VM reduces its dependence on the correctness of the host OS. For example, pulling the network stack into user-mode eliminates reliance on the kernel's network infrastructure. Similarly, a FUSE file system that directly interacts with a block device can avoid many kernel paths involving VFS and the host's native file system.

Alto can also decrease the number of trusted kernel paths by preventing VMs from triggering certain system calls. Consider system calls like `ptrace()`, `clone()`, and `ioctl()`. These system calls are powerful, with the ability to create, destroy, or tamper with process state and device state. These system calls also have complex parameters, which means that these system calls have an abundance of corner cases in which security vulnerabilities may hide. For example, Linux defines 635 `ioctl()` operation codes; however, only 52 of them are commonly used [45]. Ideally, the Alto managed runtime would export an interface that is expressive, but whose implementation only requires commonly-used system call paths that are likely to be heavily audited (and thus more secure [28]). Identifying that set of paths is a core research challenge. Earlier work has identified subsets of system calls that are rarely needed [7, 45]. However, earlier work has not identified a set of system calls that is *expressive enough to create a reasonable managed interface*, and also *likely to be secure*. Part of Alto's research contribution will be to identify such a subset.

Determining the minimal syscall interface needed to support a security-conscious managed runtime is related to, but different than, prior investigations of minimal interfaces for supporting full POSIX/Win32 semantics [22, 42]. Ultimately, our investigation may reveal that Alto VMs should run atop a custom Alto hypervisor that does not implement full POSIX/Win32 semantics, and only strives to support Alto VMs.

Challenge 5: Can Alto support hundreds of thousands of VMs per host? Assuming that Alto is successful in producing very small VMs, memory pressure on the host will not be the fundamental barrier to running many simultaneous VMs. Instead, the difficulties will involve *scheduling* and *IO efficiency*. Both difficulties relate to the manner in which the Alto runtime multiplexes VMs atop OS-level resources. For example, if Alto maps each VM to an OS-level process, then the scheduling problem is entirely an OS problem. Alternatively, Alto could employ multiple user-level threads inside a single OS-level process, with each thread running a separate event loop for a different VM from the same tenant. Regardless, the Alto runtime will have to use a system call like `epoll()` to watch for activity on a large number of file descriptors. At Alto's target load of hundreds of thousands of simultaneous VMs, the performance bottleneck may become `epoll()` itself (or another part of the kernel's IO subsystem). Empirical work is needed to identify such bottlenecks, and propose kernel-level solutions if necessary.

4 CONCLUSION

In this paper, we propose Alto, a new virtualization technique for datacenter operators. Alto's goal is to provide VMs that are small, secure, and strongly isolated. To achieve this goal, Alto's high-level approach is to virtualize at the managed runtime layer instead of the hardware level like Xen, or the POSIX layer like Docker. To enable memory-compact VMs, Alto will use trace-driven, site-specific allocators, and garbage collectors that prize high-density liveness. To support efficient snapshotting, migration, and resumption of VMs, Alto will hoist core system state, traditionally scattered across kernel memory and user memory, wholly into user memory; Alto will also make each piece of high-level VM state explicitly enumerable and addressable, allowing datacenter operators to easily migrate or patch an individual VM or a group of related VMs. To improve security, Alto will identify a minimal number of host-level system calls which must be trusted to implement the richer Alto runtime that is exposed to VMs. Various research challenges remain to be solved, but if Alto is successful, it will allow datacenter operators to efficiently and securely execute hundreds of thousands of VMs on a single physical machine.

REFERENCES

[1] O. Agesen, A. Garthwaite, J. Sheldon, and P. Subrahmanyam. 2010. The Evolution of an x86 Virtual Machine Monitor. *SIGOPS Operating Systems Review* 44, 4 (December 2010), 3–18.

[2] F. Akita. 2017. Windows Subsystem For Linux Is Good, But Not Enough Yet. (September 20, 2017). Akita On Rails Blog. http://www.akitaonrails.com/2017/09/20/windows-subsystem-for-linux-is-good-but-not-enough-yet.

[3] Amazon. 2017. AWS Lambda. (2017). https://aws.amazon.com/lambda/.

[4] Apache Software Foundation. 2017. Apache OpenWhisk. (2017). https://openwhisk.apache.org/.

[5] Apache Software Foundation. 2018. Kafka: A Distributed Streaming Platform. (2018). https://kafka.apache.org/.

[6] J. Arnold and M. F. Kaashoek. 2009. Ksplice: Automatic Rebootless Kernel Updates. In *Proceedings of EuroSys*.

[7] V. Atlidakis, J. Andrus, R. Geambasu, D. Mitropoulos, and J. Nieh. 2016. POSIX Abstractions in Modern Operating Systems: The Old, the New, and the Missing. In *Proceedings of EuroSys*.

[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. 2003. Xen and the Art of Virtualization. In *Proceedings of SOSP*.

[9] N. Brown. 2014. Control Groups Series. (July 7, 2014). Linux Weekly News. https://lwn.net/Articles/604609/.

[10] A. Cangiano. 2013. ruby-benchmark-suite: A set of Ruby benchmarks for testing Ruby implementations. (2013). https://github.com/acangiano/ruby-benchmark-suite.

[11] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. 2005. Live Migration of Virtual Machines. In *Proceedings of NSDI*.

[12] D. Clifford, H. Payer, M. Stanton, and B. L. Titzer. 2015. Memento Mori: Dynamic Allocation-Site-Based Optimizations. In *Proceedings of ACM ISMM*.

[13] CRIU. 2017. Checkpoint/Restore Tool. (October 22 2017). https://github.com/checkpoint-restore/criu.

[14] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. 2008. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proceedings of NSDI*.

[15] J. Dean and L. Barraso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (February 2013).

[16] Docker. 2017. Docker Overview. (2017). https://docs.docker.com/engine/docker-overview/.

[17] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan. 2012. High performance network virtualization with SR-IOV. *Parallel and Distributed Computing* 72, 11 (November 2012), 1471–1480.

[18] EU Parliament. 2017. GDPR Portal. (2017). http://www.eugdpr.org/eugdpr.org.html.

[19] FUSE. 2018. Reference implementation of the Linux FUSE (Filesystem in Userspace) interface. (2018). https://github.com/libfuse/libfuse.

[20] Google. 2017. Cloud Functions: A serverless environment to build and connect cloud services. (2017). https://cloud.google.com/functions/.

[21] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. In *Proceedings of HotCloud*.

[22] J. Howell, B. Parno, and J. R. Douceur. 2013. Embassies: Radically Refactoring the Web. In *Proceedings of NSDI*.

[23] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. 2013. Speeding up Distributed Request-Response Workflows. In *Proceedings of SIGCOMM*.

[24] EunYoung Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *Proc. 11th USENIX NSDI*.

[25] A. Kadav and M. Swift. 2009. Live Migration of Direct-access Devices. *SIGOPS Operating Systems Review* 43, 3 (July 2009), 95–104.

[26] M. Kerrisk. 2013. Namespaces in Operation, Part 1: Namespaces Overview. (January 4, 2013). Linux Weekly News. https://lwn.net/Articles/531114/.

[27] G. Lawrence. 2016. Dirty COW (CVE-2016-5195): Docker Container Escape. (October 26, 2016). Paranoid Software Blog. https://blog.paranoidsoftware.com/dirty-cow-cve-2016-5195-docker-container-escape/.

[28] Y. Li, B. Dolan-Gavitt, S. Weber, and J. Cappos. 2017. Lock-in-Pop: Securing Privileged Operating System Kernels by Keeping on the Beaten Path. In *Proceedings of USENIX ATC*.

[29] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of NSDI*.

[30] Linux Foundation. 2018. Data Plane Development Kit (DPDK). (2018). http://dpdk.org/.

[31] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam, J. Crowcroft, and I. Leslie. 2015. Jitsu: Just-In-Time Summoning of Unikernels. In *Proceedings of NSDI*.

[32] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. 2013. Unikernels: Library Operating Systems

for the Cloud. In *Proceedings of ASPLOS*.

[33] J. Martins, M. Ahmed, C. Raiciu, and F. Huici. 2013. Enabling Fast, Dynamic Network Processing with ClickOS. In *Proceedings of HotSDN*.

[34] Microsoft. 2017. Hyper-V Architecture. (2017). https://msdn.microsoft.com/en-us/library/cc768520(v=bts.10).aspx.

[35] D. Morsing. 2014. How Stacks are Handled in Go. (September 15, 2014). CloudFlare. https://blog.cloudflare.com/how-stacks-are-handled-in-go/.

[36] I. Neamtiu, M. Hicks, J. S. Foster, and P. Pratikakis. 2008. Contextual Effects for Version-Consistent Dynamic Software Updating and Safe Concurrent Programming. In *Proceedings of POPL*.

[37] M. Nelson, B.-H. Lim, and G. Hutchins. 2005. Fast Transparent Migration for Virtual Machines. In *Proceedings of USENIX ATC*.

[38] OpenAPI Initiative. 2018. OpenAPI Specification Repository. (2018). https://github.com/OAI/OpenAPI-Specification.

[39] Z. Pan, Y. Dong, Y. Chen, L. Zhang, and Z. Zhang. 2012. CompSC: Live Migration with Pass-through Devices. In *Proceedings of VEE*.

[40] Pivotal. 2018. RabbitMQ: Messaging That Just Works. (2018). https://www.rabbitmq.com/.

[41] G. Polito, S. Ducasse, L. Fabresse, and N. Bouraqadi. 2013. Virtual Smalltalk Images: Model and Applications. In *Proceedings of the International Smalltalk Conference*.

[42] D. E. Porter, S. Boyd-Wickizer, J.Howell, R. Olinsky, and G. C. Hunt. 2011. Rethinking the Library OS from the Top Down. In *Proceedings of ASPLOS*.

[43] J. R. Santos, Y. Turner, G. Janakiraman, and I. Pratt. 2008. Bridging the Gap between Software and Hardware Techniques for I/O Virtualization. In *Proceedings of USENIX ATC*.

[44] D. Shapira. 2017. Escaping Docker container using waitid(): CVE-2017-5123. (December 27, 2017). Twistlock. https://www.twistlock.com/2017/12/27/escaping-docker-container-using-waitid-cve-2017-5123/.

[45] C.-C. Tsai, B. Jain, N. A. Abdul, and D. E. Porter. 2016. A Study of Modern Linux API Usage and Compatibility: What to Support When YouâĂŹre Supporting. In *Proceedings of EuroSys*.

[46] VMware. 2017. Configuration Maximums: vSphere 6.0. (2017). https://www.vmware.com/pdf/vsphere6/r60/vsphere-60-configuration-maximums.pdf.

[47] T. Wagner. 2014. Understanding Container Reuse in AWS Lambda. (December 31, 2014). AWS Compute Blog. https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/.

[48] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. 2013. Bobtail: Avoiding Long Tails in the Cloud. In *Proceedings of NSDI*.

[49] A. Yigal. 2017. Should You Go "Serverless"? The Pros and Cons. (January 2, 2017). https://devops.com/go-serverless-pros-cons/.