# Isolating Functions at the Hardware Limit with Virtines

Nicholas C. Wanninger*
ncw@u.northwestern.edu
Northwestern University
Evanston, Illinois, USA

Joshua J. Bowden
jbowden@hawk.iit.edu
Illinois Institute of Technology
Chicago, Illinois, USA

Kirtankumar Shetty
kshetty11@hawk.iit.edu
Illinois Institute of Technology
Chicago, Illinois, USA

Ayush Garg
agarg22@hawk.iit.edu
Illinois Institute of Technology
Chicago, Illinois, USA

Kyle C. Hale
khale@cs.iit.edu
Illinois Institute of Technology
Chicago, Illinois, USA

## Abstract

An important class of applications, including programs that leverage third-party libraries, programs that use user-defined functions in databases, and serverless applications, benefit from isolating the execution of untrusted code at the granularity of individual functions or function invocations. However, existing isolation mechanisms were not designed for this use case; rather, they have been adapted to it. We introduce *virtines*, a new abstraction designed specifically for function granularity isolation, and describe how we build virtines from the ground up by pushing hardware virtualization to its limits. Virtines give developers fine-grained control in deciding which functions should run in isolated environments, and which should not. The virtine abstraction is a general one, and we demonstrate a prototype that adds extensions to the C language. We present a detailed analysis of the overheads of running individual functions in isolated VMs, and guided by those findings, we present Wasp, an embeddable hypervisor that allows programmers to easily use virtines. We describe several representative scenarios that employ individual function isolation, and demonstrate that virtines can be applied in these scenarios with only a few lines of changes to existing codebases and with acceptable slowdowns.

***CCS Concepts:*** • **Security and privacy** → *Virtualization and security*; • **Software and its engineering** → *Software architectures*; Language features; *Runtime environments*.

*A majority of this work was done while at Illinois Institute of Technology.



*EuroSys '22, April 5–8, 2022, RENNES, France*



ACM ISBN 978-1-4503-9162-7/22/04…$15.00
https://doi.org/10.1145/3492321.3519553

***Keywords:*** virtines, virtualization, isolation

**ACM Reference Format:**
Nicholas C. Wanninger, Joshua J. Bowden, Kirtankumar Shetty, Ayush Garg, and Kyle C. Hale. 2022. Isolating Functions at the Hardware Limit with Virtines. In *Seventeenth European Conference on Computer Systems (EuroSys '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3492321.3519553

## 1 Introduction

Vulnerabilities in critical applications can lead to information leakage, data corruption, control-flow hijacking, and other malicious activity. If vulnerable applications run with elevated privileges, the entire system may be compromised [4, 7]. Systems that execute code from untrusted sources must then employ isolation mechanisms to ensure data secrecy, data integrity, and execution integrity for critical software infrastructure [20, 25, 27, 31–33, 51, 56–58, 71, 72]. This isolation typically happens in a coarse-grained fashion, but an important class of applications require isolation at the granularity of individual functions or distinct invocations of such functions. Long-standing examples include the use of untrusted library functions by critical applications and user-defined functions (UDFs) in databases, while serverless functions represent an important emerging example. Existing isolation mechanisms, however, were not designed for individual functions. Applications that leverage this isolation model must resort to repurposing off-the-shelf mechanisms with mismatched design goals to suit their needs. For example, databases limit UDFs to run in a managed language like Java or Javascript [29, 63], and serverless platforms repurpose containers to isolate users' stateless function invocations from one another. The latter example is particularly salient today. As others have shown at this venue, formidable challenges (particularly cold-start latency) arise when using containers for individual function execution [21]. These challenges stem from contorting the container abstraction to fit an unintended usage model.

Guided by these examples, we introduce *virtines*, a new abstraction *designed* for isolating execution at function call granularity using hardware virtualization. Data touched by

a virtine is automatically encapsulated in the virtine's isolated execution environment. This environment implements an abstract machine model that is not constrained by the traditional x86 platform. Virtines can seamlessly interact with the host environment through a checked hypervisor interposition layer. With virtines, programmers annotate critical functions in their code using language extensions, with the semantics that a single virtine will run in its own, isolated virtual machine environment. While virtines require code changes, these changes are minimal and easy to understand. Our current language extensions are for C, but we believe they can be adapted to most languages.

The execution environments for virtines (including parts of the hypervisor) are tailored to the code inside the isolated functions; a virtine image contains *only* the software that a function needs. We present a detailed, ground-up analysis of the start-up costs for virtine execution environments, and apply our findings to construct small and efficient virtine images. Virtines can achieve isolated execution microsecond scale startup latencies and limited slow-down relative to native execution. They are supported by a custom, user-space runtime system implemented using hardware virtualization called Wasp, which comprises a small, embeddable hypervisor that runs on both Linux and Windows. The Wasp runtime provides mechanisms to enforce strong virtine isolation by default, but isolation policies can be customized by users.

Our contributions in this paper are as follows:

- We introduce *virtines*, programmer-guided abstractions that allow individual functions to run in light-weight, virtualized execution environments.
- We present a prototype embeddable hypervisor framework, Wasp, that implements the virtine abstraction. Wasp runs as a Type-II micro-hypervisor on both Linux and Windows.
- We provide language extensions for programming with virtines in C that are conceptually simple.
- We evaluate Wasp's performance using extensive microbenchmarking, and perform a detailed study of the costs of virtine execution environments.
- We demonstrate that it requires minimal effort to incorporate virtines into software components used in representative scenarios involving function isolation: namely, untrusted or sensitive library functions (OpenSSL) and managed language runtimes (Javascript). The virtine versions incur acceptable slow-down while using strong hardware isolation.

## 2 Virtines

A *virtine* provides an isolated execution environment using lightweight virtualization. Virtines consist of three components: a toolchain-generated binary to run in a virtual context, a hypervisor that facilitates the VM's only external access (Wasp), and a host program which specifies virtine isolation policies and drives Wasp to create virtines. When invoked, virtines run synchronously from the caller's perspective, leading them to appear and act like a regular function invocation. However, virtines could, given support in the hypervisor, behave like asynchronous functions or futures.[1] As with most code written to execute in a different environment from the host, such as CUDA code [59] or SGX enclaves [69], there are constraints on what virtine code can and cannot do. Due to their isolated nature, virtines have no direct access to the caller's environment (global variables, heap, etc.). A virtine can, however, accept arguments and produce return values like any normal function. These arguments and return values are marshalled automatically by the virtine compiler when using our language extensions.[2]

Unlike traditional hypervisors, a virtine hypervisor need not–and we suspect in most cases will not–emulate every part of the x86 platform, such as PCI, ACPI, interrupts, or legacy I/O. A virtine hypervisor therefore implements an abstract machine model designed for and restricted to the intentions of the virtine. Figure 1 outlines the architecture and data access capabilities (indicated by the arrows) of a virtine compared to a traditional process abstraction. A host program that uses (links against) the embeddable virtine hypervisor has some, but not necessarily all, of its functions run as virtines. We refer to such a host process as a virtine *client*. If the virtine context wishes to access any data or service outside of its isolated environment, it must first request access from the client via the hypervisor. Virtines exist in a default-deny environment, so the hypervisor must interpose on all such requests. While the hypervisor provides the interposition mechanism [17], the virtine client has the option to implement a hypercall policy, which determines whether or not an individual request will be serviced. The capabilities of a virtine are determined by (1) the hypervisor, (2) the runtime within a virtine image, and (3) policies determined by the virtine client.

Virtines are constructed from a subset of an application's call graph. Currently, the decision where the "cut" in the call graph is made by the programmer, but making this choice automatically in the compiler is possible [46, 51]. Since a virtine constitutes only a subset of the call graph, virtine images are typically small (~16KB), and are statically compiled binaries containing all required software. Shared libraries violate our isolation requirements, as we will see in Section 3.1.

While the runtime environment that underlies a function running in virtine context can vary, we expect that in most cases this environment will comprise a limited, kernel-mode only, software layer. This may mean no scheduler, virtual

[1] For example, like *goroutines*, as in Gotee [28]: https://gobyexample.com/goroutines

[2] When using the virtine runtime library directly, developers must currently marshal arguments and return values manually, though we are currently developing an IDL to ease this process (like SGX's EDL [36]).

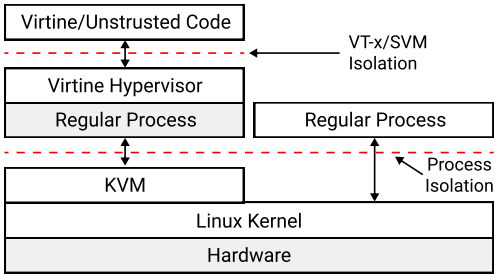


**Figure 1.** How virtines fit into the software stack.

memory, processes, threads, file systems, or any other high-level constructs that typically come with running a fully-featured VM. This is not, however, a requirement, and virtines can take advantage of hardware features like virtual memory, which can lead to interesting optimizations like those in Dune [18]. Additional functionality *must* be provided by adding the functionality to the virtine environment or by borrowing functionality from the hypervisor. Adding this functionality should be done with care, as interactions with the hypervisor come with costs, both in terms of performance and isolation. In this paper, we provide two pre-built virtine execution environments (Section 5.4), but we envision a rich virtine ecosystem could develop from which an execution environment could be selected. These environments could also be synthesized automatically. Note that one possible execution environment for a virtine is a unikernel. However, unikernels are typically designed with a standard ABI in mind (e.g., binary compatible with Linux). Virtine execution environments are instead co-designed with the virtine client, and allow for a wide variety of virtual platforms which may support non-standard ABIs.

# 3 Design

In this section, we describe our isolation and safety objectives in developing the virtine abstraction. We then discuss how to achieve these goals using hardware and software mechanisms.

## 3.1 Safety Objectives

***Host execution and data integrity.*** Host code and data cannot be modified, and its control flow cannot be hijacked by a virtine running untrusted or adversarial code.

***Virtine execution and data integrity.*** The private state of a virtine must not be affected by another virtine running untrusted or adversarial code. Thus, data secrecy must be maintained between virtines.

***Virtine isolation.*** Host data secrecy must also be maintained, so virtines may not interact with any data or services outside of their own address space other than what is explicitly permitted by the virtine client's policies.

These objectives are similar to sandboxing in web browsers, where some components (tabs, extensions) within the same address space are untrusted, meaning that intra-application interactions must cross isolation boundaries. In Google Chrome, for example, process isolation and traditional security restrictions are used to achieve this isolation [1, 50]. Unfortunately, as with most software, bugs have allowed attackers to access user data, execute arbitrary code, or just crash the browser with carefully crafted JavaScript [2, 3, 5, 6, 8].

## 3.2 Threat Model

Code that runs in virtine context can still suffer from software bugs such as buffer overflow vulnerabilities. We therefore assume an adversarial model, where attacks that arise from such bugs may occur, and where a virtine can behave maliciously. We assume the hypervisor (Wasp) and the host kernel are trusted, similar to prior work [14]. In addition, we assume that the virtine client—in particular, its hypercall handlers—are trusted and implemented correctly. These handlers must take care to assume that inputs have *not* been properly sanitized, and may even be intentionally manipulated. Along with using best practices, we assume hypercall handlers are careful when accessing the resources mapped to a virtine, for example checking memory bounds before accessing virtine memory, validating potentially unsafe arguments, and correctly following the access model that the virtine requires. We assume that virtines do not share state with each other via shared mappings, and that they cannot directly access host memory. Additionally, we assume that microarchitectural and host kernel mitigations are sufficient to eliminate side channel attacks. Note that we do not expect end users to implement their own virtine clients. We instead assume that runtime experts will develop the virtine clients (and corresponding hypercall handlers). In this sense, our assumptions about the virtine client's integrity are similar to those made in cloud platforms that employ a user-space device model (e.g., QEMU/KVM).

## 3.3 Achieving Safety Objectives

***Host execution integrity.*** By assuming that both the hypervisor and client-defined hypercall handlers (of which there are few) are carefully implemented, using best practices of software development, an adversarial virtine *cannot* directly modify the state or code paths of the host. However, virtines *do not* guarantee that if permitted access to certain hypercalls or secret data, an attacker cannot utilize these hypercalls to exfiltrate sensitive data via side-channel mechanisms. This, however, can be mitigated by using a mechanism that disables certain hypercalls dynamically when they are not needed by the runtime, further restricting the attack surface.

***Virtine execution integrity.*** Requiring that no two virtines directly share memory without first receiving permission from the hypervisor (e.g., via the hypercall interface) ensures data

secrecy within the virtine. Each virtine must have its own set of private data which must be disjoint from any other virtine's set. Thus, a virtine that runs untrusted or malicious code cannot affect the integrity of other virtines.

***Isolation.*** Modeling virtine and host private state as a disjoint set disallows any and all shared state between virtines or the host. The hardware's use of nested paging (EPT in VT-x) prevents such access at a hardware level. Also, by assuming that hypercalls are carefully implemented, and that they only permit operations required by the application, we achieve isolation from states and services outside the virtine.

# 4 Minimizing Virtine Costs

Before exploring the implementation of virtines, we first describe a series of experiments that guided their design. These experiments establish the creation costs of minimal virtual contexts and of the execution environments used within those contexts. Our goal is to establish what forms of overhead will be most significant when creating a virtine.

## 4.1 Experimental Setup

The majority of our Linux and KVM experiments were run on *tinker*, an AMD EPYC 7281 (Naples; 16 cores; 2.69 GHz) machine with 32 GB DDR4 running stock Linux kernel version 5.9.12. We disabled hyperthreading, turbo boost, and DVFS to mitigate measurement noise. We used a Dell XPS 9500 with an Intel i7 10750H (Comet Lake; 6 cores) for SGX measurements. This machine has 32 GB DDR4 and runs stock Ubuntu 20.04 (kernel version 5.13.0-28). We used gcc 10.2.1 to compile Wasp (C/C++), clang 10.0.1 for our C-based virtine language extensions, and NASM v2.14 for assembly-only virtines. Unless otherwise noted, we conduct experiments with 1000 trials. Note that our hypervisor implementation works on both Linux and has a prototype implementation in Windows (through Hyper-V), but for brevity we only show KVM's performance on Linux, as Hyper-V performance was similar for our experiments.

## 4.2 Measuring Startup Costs

We probe the costs of virtual execution contexts and see how they compare to other types of execution contexts. To establish baseline creation costs, we measure how quickly various execution contexts can be constructed on *tinker*, as shown in Figure 2. We measure the time it takes to create, enter, and exit from the context in a way that the hypervisor can observe. In "KVM", we observe the latency to construct a virtual machine and call the `hlt` instruction. "Linux pthread" is simply a `pthread_create` call followed by `pthread_join`. The "vmrun" measurement is the cost of running a VM hosted on KVM without the cost of creating its associated state, i.e., only the `KVM_RUN ioctl`. Finally, "function" is the cost of calling and returning from a null function. All measurements are obtained using the `rdtsc` instruction.

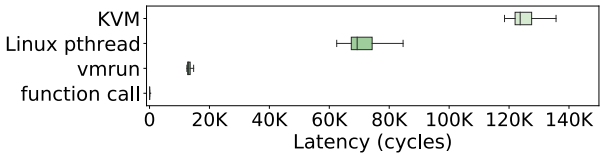


**Figure 2.** Lower bounds on execution context creation in cycles (measured with `rdtsc`).

***Lower bounds.*** The "vmrun" measurements represent the lowest latency we could achieve to begin execution in a virtual context using KVM in Linux 5.9. This latency includes the cost of the `ioctl` system call, which in KVM is handled with a series of sanity checks followed by execution of the `vmrun` instruction. Several optimizations can be made to the hypervisor to reduce the cost of spawning new contexts and lower the latency of a virtine, which we outline in Section 5.2.

These measurements tell us that while a virtine invocation will be unsurprisingly more expensive than a native function call, it can compete with thread creation and will far outstrip any start-up performance that processes (and by proxy, containers) will achieve in a standard Linux setting. We conclude that the baseline cost of *creating* a virtual context is relatively inexpensive compared to the cost of other abstractions.

***Eliminating traditional boot sequences.*** The boot sequences of fully-featured OSes are too costly to include on the critical path for low-latency function invocations [21, 41, 55]. It takes hundreds of milliseconds to boot a standard Linux VM using QEMU/KVM. To understand why, we measured the time taken for components of a vanilla Linux kernel boot sequence and found that roughly 30% of the boot process is spent scanning ACPI tables, configuring ACPI, enumerating PCI devices, and populating the root file system. Most of these features, such as a fully-featured PCI interface, or a network stack, are unnecessary for short-lived, virtual execution environments, and are often omitted from optimized Linux guest images such as the Alpine Linux image used for Amazon's Firecracker [13]. Caching pre-booted environments can further mitigate this overhead, as we describe in §5.2.

In light of the data gathered in Figure 2, we set out to measure the cost of creating a virtual context and configuring it with the fewest operations possible. To do this, we built a simple wrapper around the KVM interface that loads a binary image compiled from roughly 160 lines of assembly. This binary closely mirrors the boot sequence of a classic OS kernel: it configures protected mode, a GDT, paging, and finally jumps to 64-bit code. These operations are outlined in Table 1, which indicates the minimum latencies (cycles) for each component, ordered by cost.

The row labeled "Paging/ident. map" is by far the most expensive at ~28K cycles. Here we are using 2MB large pages to identity map the first 1GB of address space, which entails

| Component | KVM |
|---|---|
| Paging identity mapping | 28109 |
| Protected transition | 3217 |
| Long transition (`lgdt`) | 681 |
| Jump to 32-bit (`ljmp`) | 175 |
| Jump to 64-bit (`ljmp`) | 190 |
| Load 32-bit GDT (`lgdt`) | 4118 |
| First Instruction | 74 |

**Table 1.** Boot time breakdown for our minimal runtime environment on KVM. These are minimum latencies observed per component, measured in cycles.

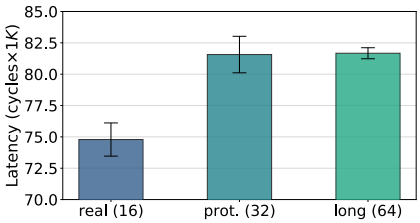


**Figure 3.** Latency to run a function in the three classic operating modes on x86. Note the use of a false origin to highlight relative differences.

three levels of page tables (i.e., 12KB of memory references), plus the actual installation of the page tables, control register configuration, and construction of an EPT inside KVM. The transition to protected mode takes the second longest, at 3K cycles. This is a bit surprising, given that this only entails the protected mode bit flip (PE, bit 0) in `cr0`. The transition to long mode (which takes several hundreds of cycles) is less significant. The remaining components—loading a 32-bit GDT, the long jumps to complete the mode transitions, and the initial interrupt disable—are negligible.

***The cost of processor modes.*** The more complex the mode of execution (16, 32, or 64 bits), the higher the latency to get there. This is consistent with descriptions in the hardware manuals [15, 37]. To further investigate this effect, we invoked a small binary written in assembly that brings the virtual context up to a particular x86 execution mode and executes a simple function (`fib` of 20 with a simple, recursive implementation). Figure 3 shows our findings for the three canonical modes of the x86 boot process using KVM: 16-bit (real) mode, 32-bit (protected) mode, and 64-bit (long) mode. Each mode includes the necessary components from Table 1 in the setup of the virtual context. In this experiment, for each mode of execution, we measured the latency in cycles from the time we initiated an entry on the host (`KVM_RUN`), to the time it took to bring the machine up to that mode in the guest (including the necessary components listed in Table 1),

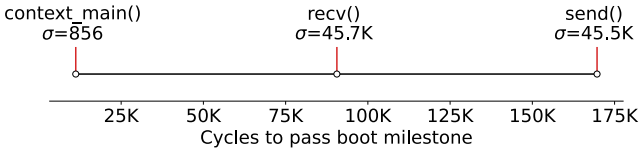


**Figure 4.** Latency for echo server startup milestones in protected mode (no paging).

run $fib(20)$, and exit back to the host. These measurements include entry, startup cost, computation, and exit. Note that we saw several outliers in all cases, likely due to host kernel scheduling events. To make the data more interpretable, we removed these outliers.[3]

While we expect much of the time to be dominated by entry/exit and the arithmetic, the benefits of real-mode only execution for our hand-written version are clear. The difference between 16-bit and 32-bit environments are not surprising. The most significant costs listed in Table 1 are not incurred when executing in 16-bit mode. Protected and Long mode execution are essentially the same as they both include those costs (paging and protected setup). These results suggest—provided that the virtine is short-lived (on the order of *microseconds*) and can feasibly execute in real-mode—that 10K cycles may potentially be saved.

***Booting up for useful code.*** We have seen that a minimal long-mode boot sequence costs less than 30K cycles (~12 $\mu$s), but what does it take to do something useful? To determine this, we implemented a simple HTTP echo server where each request is handled in a new virtual context employing our minimal environment. We built a simple micro-hypervisor in C++ and a runtime environment that brings the machine up to C code and uses hypercall-based I/O to echo HTTP requests back to the sender. The runtime environment comprises 970 lines of C (a large portion of which are string formatting routines) and 150 lines of x86 assembly. The micro-hypervisor comprises 900 lines of C++. The hypercall-based I/O (described more in Section 5.1) obviates the need to emulate network devices in the micro-hypervisor and implement the associated drivers in the virtual runtime environment, simplifying the development process. Figure 4 shows the mean time measured in cycles to pass important startup milestones during the bring-up of the server context. The left-most point indicates the time taken to reach the server context's main entry point (C code); roughly 10K cycles. Note that this example does not actually require 64-bit mode, so we omit paging and leave the context in protected mode. The middle point shows the time to receive a request (the return from `recv()`), and the last point shows the time to complete the response (`send()`). Milestone measurements are taken inside the virtual context.

[3]That is, using Tukey's method, measurements not on the interval $[x_{25\%} - 1.5\,IQR, x_{75\%} + 1.5\,IQR]$ are removed from the data.

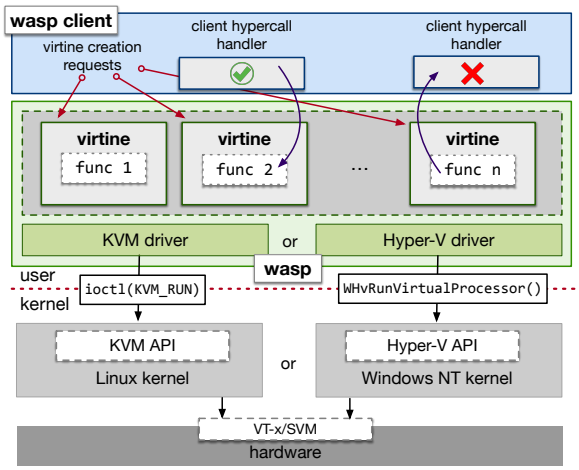


**Figure 5.** High-level overview of Wasp.

The send and receive functions for this environment use hypercalls to defer to the hypervisor, which proxies them to the Linux host kernel using the appropriate system calls. Even when leveraging the underlying host OS, and when adding the from-scratch virtual context creation time from Figure 2, we can achieve sub-millisecond HTTP response latencies (<300 $\mu$s) *without optimizations* (§5.2). Thus, we can infer that despite the cost of creating a virtual context, having few host/virtine interactions can keep execution latencies in a virtual context within an acceptable range. Note, however, that the guest-to-host interactions in this test introduce variance from the host kernel's network stack, indicated by the large standard deviations shown in Figure 4.

These results are promising, and they indicate that we can achieve low overheads and start-up latencies for functions that do not require a heavy-weight runtime environment. We use three key insights from this section to inform the design of our virtine framework in the next section: (1) creating hardware virtualized contexts can be cheap when the environment is small, (2) tailoring the execution environment (for example, the processor mode) can pay off, and (3) host interactions can be facilitated with hypercalls (rather than shared memory), but their number must be limited to keep costs low.

# 5 Implementation

In this section, we present Wasp, a prototype hypervisor designed for the creation and management of virtine environments. We also cover a few of the optimizations designed to overcome the cost of creating virtual contexts using KVM.

## 5.1 Wasp

Wasp is a specialized, embeddable micro-hypervisor runtime that deploys virtines with an easy-to-use interface. Wasp runs on Linux and Windows. At its core, Wasp is a fairly ordinary hypervisor, hosting many virtual contexts on top of a host OS. However, like other minimal hypervisors such as Firecracker [13], Unikernel monitors [75], and uhyve [45], Wasp does not aim to emulate the entire x86 platform or device model. As shown in Figure 5, Wasp is a userspace runtime system built as a library that host programs (virtine clients) can link against. Wasp mediates virtine interactions with the host via a hypercall interface, which is checked by the hypervisor and the virtine client. The figure shows one virtine that has no host interactions, one virtine which makes a valid hypercall request, and another whose hypercall request is denied by the client-specified security policy. By using Wasp's runtime API, a virtine client can leverage hardware specific virtualization features without knowing their details. Several types of applications (including dynamic compilers and other runtime systems) can link with the Wasp runtime library to leverage virtines. On Linux, each virtual context is represented by a device file which is manipulated by Wasp using an `ioctl`.

Wasp provides no libraries to the binary being run, meaning they have no in-virtine runtime support by default. Wasp simply accepts a binary image, loads it at guest virtual address `0x8000`, and enters the VM context. Any extra functionality must be achieved by interacting with the hypervisor and virtine client. In Wasp, delegation to the client is achieved with hypercalls using virtual I/O ports.

Hypercalls in Wasp are not meant to emulate low-level virtual devices, but are instead designed to provide high-level hypervisor services with as few exits as possible. For example, rather than performing file I/O by ultimately interacting with a virtio device [67] and parsing filesystem structures, a virtine could use a hypercall that mirrors the `read` POSIX system call. Hypercalls vector to a co-designed handler either provided by Wasp or implemented by the virtine client. Wasp provides the mechanisms to create virtines, while the client can specify security policies through handlers. These handlers could simply run a series of checks and pass through certain host system calls while filtering others out. While virtine clients can implement custom hypercall handlers, they can also choose from a variety of general-purpose handlers that Wasp provides out-of-the-box; these canned hypercalls are used by our language extensions (§5.3). By default, Wasp provides no externally observable behavior through hypercalls other than the ability to exit the virtual context; all other external behavior must be validated and expressly permitted by the custom (or canned) hypercall handlers, which are implemented (or selected) by the virtine client.

## 5.2 Wasp Caching and Snapshotting

***Caching.*** To reduce virtine start-up latencies, Wasp supports a pool of cached, uninitialized, virtines (shells) that can be reused. As depicted in Figure 6, Wasp receives a request from a virtine client (A), which will drive virtine creation. Such requests can be generated in a variety of virtine client scenarios. For example, network traffic hitting a web server

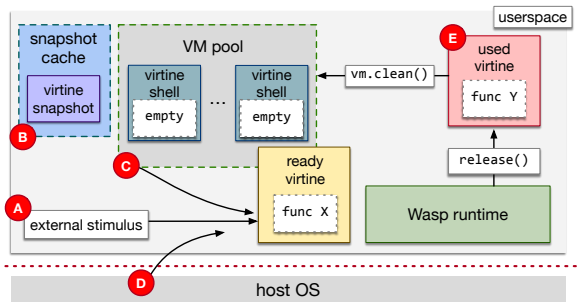


**Figure 6.** Image snapshotting and virtine reuse with a pooled design.

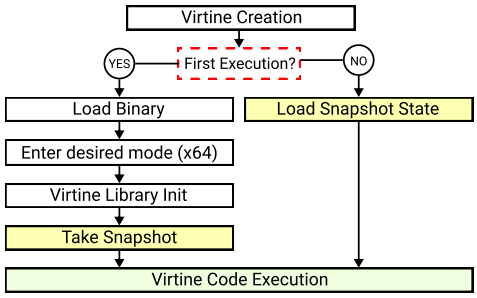


**Figure 7.** Virtine code path on first execution and subsequent executions after snapshotting.

that implements a virtine client may generate virtine invocations. A database engine incorporating a virtine client may run virtine-based UDFs in response to triggers. Because we must use a new virtine for every request, a hardware virtual context must be provisioned to handle each invocation. The context is acquired by one of two methods, provisioning a clean virtual context (C) or reusing a previously created context (D). When the system is cold (no virtines have yet been created), we must ask the host kernel for a new virtual context by using KVM's `KVM_CREATE_VM` interface. If this route is taken, we pay a higher cost to construct a virtine due to the host kernel's internal allocation of the VM state (VMCS on Intel/VMCB on AMD). However, once we do this, and the relevant virtine returns, we can clear its context (E), preventing information leakage, and cache it in a pool of "clean" virtines (C) so the host OS need not pay the expensive cost of re-allocating virtual hardware contexts. These virtine "shells" sit dormant waiting for new virtine creation requests (B). The benefits of pooling virtines are apparent in Figure 8 by comparing creation of a Wasp virtine from scratch (the "Wasp" measurement) with reuse of a cached virtine shell from the pool ("Wasp+C"). By recycling virtines, we can reach latencies much lower than Linux thread creation and much closer to the hardware limit, i.e., the `vmrun` instruction. Note that here we include Linux process creation latencies as well for scale. Included is the "Wasp+CA" (cached, asynchronous) measurement, which does not measure the cost of cleaning virtines and instead cleans them asynchronously in the background. This can be implemented by either a background thread or can be done when there are no incoming requests. This measurement shows that the caching mechanism brings the cost of provisioning a virtine shell to within $4\%$ of a bare `vmrun`.

We also measured these costs on a recent SGX-enabled Intel platform and observed similar behavior, as shown in the bottom half of Figure 8. The "SGX Create" measurement indicates the cost of creating a new enclave, and the ECALL measurement indicates the cost of *entering* an enclave, thus reusing the previously created context.

***Snapshotting.*** As was shown in Section 4, the initialization of a virtine's execution state can lead to significant overheads compared to traditional function calls. This overhead is undesirable if the code that is executed in a virtine is not particularly long-lived (less than a few microseconds). Others have mitigated these start-up latencies in the serverless domain by "checkpointing" or "snapshotting" container runtime state after initialization [21, 26, 60]. In a similar fashion, Wasp supports snapshotting by allowing a virtine to leverage the work done by previous executions of the same function. As outlined in Figure 7, the first execution of a virtine must still go through the initialization process by entering the desired mode and initializing any runtime libraries (in this case, libc). The virtine then takes a snapshot of its state, and continues executing. Subsequent executions of the same virtine can then begin execution at the snapshot point and skip the initialization process. This optimization significantly reduces virtine overheads, which we explore further in Section 5.3. Of course, by snapshotting a virtine's private state, that state is exposed to all future virtines that are created using that "reset state." Thus, care must be taken in describing what memory is saved in a snapshot in order to maintain the isolation objectives outlined in Section 3.3. We detail the costs involved in snapshotting in Section 6.2.

### 5.3 C Language Extensions

While Wasp significantly eases the development and deployment of virtines, with only the runtime library, developers must still manage virtine internals, namely the build process for the virtine's internal execution environment. Requiring developers to create kernel-style build systems that package boot code, address space configurations, a minimal libc, and a linker script per virtine creates an undue burden. To alleviate this burden, we implemented a clang wrapper and LLVM compiler pass. The purpose of the clang wrapper is to include our pass in the invocation of the middle-end. The compiler pass detects C functions annotated with the `virtine` keyword, runs middle-end analysis at the IR level, and automatically generates code that invokes a pre-compiled virtine

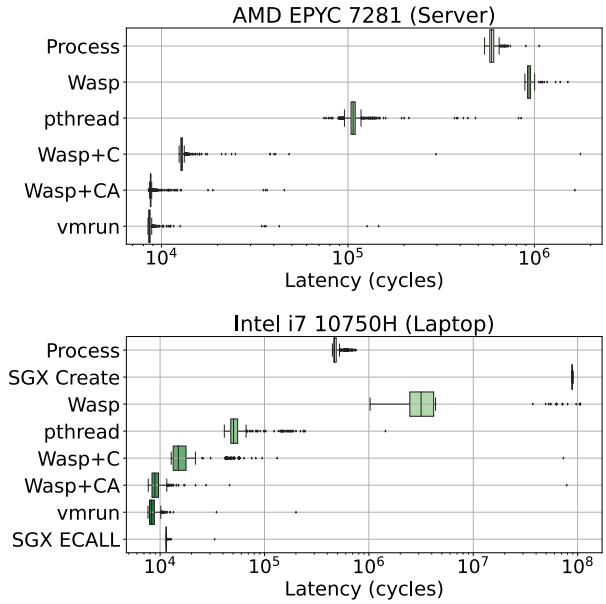


**Figure 8.** Creation latencies for execution contexts on modern AMD and Intel platforms, including Wasp virtines and SGX where available. Note the log scale on the horizontal axis.

```
virtine int fib(int n) {
    if (n < 2) return n;
    return fib(n - 1) + fib(n - 2);
}
```

**Figure 9.** Virtine programming in C with compiler support.

binary whenever the function is called. When this pass detects a function annotation as shown in Figure 9, it generates a call graph rooted at that function. The compiler automatically packages a subset of the source program into the virtine context based on what that virtine needs. Global variables accessed by the virtine are currently initialized with a snapshot when the virtine is invoked. Concurrent modifications (e.g., by different virtines, or by the client and a virtine) will occur on distinct copies of the variable. Currently, if a virtine calls another virtine-annotated function, a nested virtine will not be created.

To further ease programming burden, compiler-supported virtines must have access to some subset of the C standard library. Due to the nature of their runtime environment, basic virtines do not include these libraries. To remedy this, we created a virtine-specific port of *newlib* [9], an embeddable C standard library that statically links and maintains a relatively small virtine image size. Newlib allows developers to provide their own system call implementations; we simply forward them to the hypervisor as a hypercall. When the `virtine` keyword is used, all hypercalls are restricted

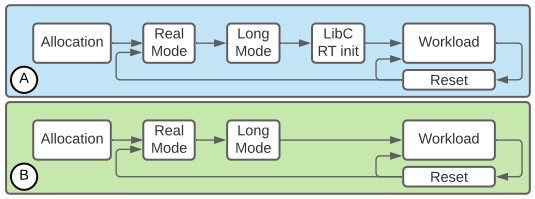


**Figure 10.** Default execution environments available to a virtine.

by default, following the default-deny semantics of virtines previously mentioned. If, however, the programmer (implementing the virtine client) would like to permit hypercalls, they can use the `virtine_permissive` keyword to allow all hypercalls, or the `virtine_config(cfg)` to supply a configuration structure that contains a bit mask of allowed hypercalls. If a hypercall is permitted, the handler in the client must validate the arguments and service it, for example by delegating to the host kernel's system call interface or by performing client-specific emulation.

This allows virtines to support standard library functionality without drastically expanding the virtine runtime environments. Of course, by using a fully fledged standard library, the user still opens themselves up to common programming errors. For example, an errant `strcpy` can still result in undefined (or malicious) behavior, but this has no consequences for the host or other virtines as outlined in Section 3.3. All virtines created via our language extensions use Wasp's snapshot feature by default. This can be disabled with the use of an environment variable.

### 5.4 Execution Environments

Wasp provides two default execution environments for programmers to use, though others are possible. These default environments are shown in Figure 10. For the C extensions (Ⓐ), the virtine is pre-packaged with a POSIX-like runtime environment, which stands between the "boot" process and the virtine's function. If a programmer directly uses the Wasp C++ API, (Ⓑ), the virtine is not automatically packaged with a runtime, and it is up to the client to provide the virtine binary. Both environments can use snapshotting after the reset stage, allowing them to skip the costly boot sequence. We envision an environment management system that will allow programmers to treat these environments much like package dependencies [49].

## 6 Evaluation

In this section, we evaluate virtines and the Wasp runtime using microbenchmarks and case studies that are representative of function isolation in the wild. With these experiments, we seek to answer the following questions:

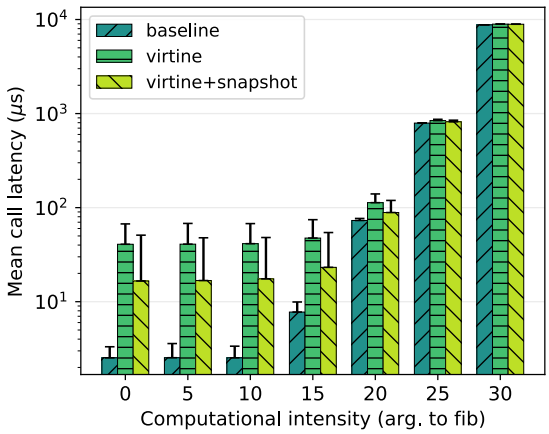


**Figure 11.** Latency of virtines as computational intensity increases. Note the log scale on the vertical axis.

- How significant are baseline virtine startup overheads with our language extensions, and how much computation is necessary to amortize them? (§6.1)
- What is the impact of the virtine's execution environment (image size) on start-up cost? (§6.2)
- What is the performance penalty for host interactions? (§6.3)
- How much effort is required to integrate virtines with off-the-shelf library code? (§6.4)
- How difficult is it to apply virtines to managed language use cases and what are the costs? (§6.5)

### 6.1 Startup Latencies with Language Extensions

We first study the start-up overheads of virtines using our language extensions. We implemented the minimal `fib` example shown in Figure 9 and scaled the argument to `fib` to increase the amount of computation per function invocation, shown in Figure 11. We compare virtines with and without image snapshotting to native function invocations. $fib(0)$ essentially measures the inherent overhead of virtine creation, and as $n$ increases, the cost of creating the virtine is amortized. The measurements include setup of a basic virtine image (which includes libc), argument marshalling, and minimal machine state initialization. The argument, $n$, is loaded into the virtine's address space at address `0x0`. In the case of the experiment labeled "virtine + snapshot," a snapshot of the virtine's execution state is taken on the first invocation of the `fib` function. All subsequent invocations of that function will use this snapshot, skipping the slow path boot sequence (see Figure 7) producing an overall speedup of 2.5× relative to virtines *without* snapshotting for $fib(0)$. Note that we are not measuring the steady state, so the bars include the overhead for taking the initial snapshot. This is why we see more variance for the snapshotting measurements.

| System | Latency | Boundary Cross Mechanism |
|---|---|---|
| Wedge [20] | ~60$\mu$s | *sthread* call |
| LwC [48] | 2.01$\mu$s | `lwSwitch` |
| Enclosures [27] | 0.9$\mu$s | Custom syscall interface |
| SeCage [51] | 0.5$\mu$s | VMRUN/VMFUNC |
| Hodor [32] | 0.1$\mu$s | VMRUN/VMFUNC |
| Virtines | 5$\mu$s | Syscall interface + VMRUN |

**Table 2.** Comparing costs of crossing isolation boundaries.

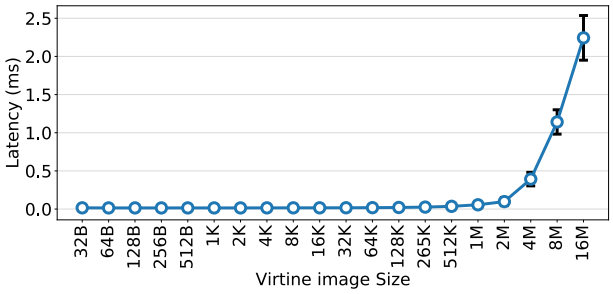


**Figure 12.** Impact of image size on start-up latency.

At first, the relative slowdown between native function invocation and virtines with snapshotting is 6.6×. When the virtine is short-lived, the costs of provisioning a virtine shell and initializing it account for most of the execution time. However, with larger computational requirements, the slowdown drops to 1.03× for $n = 25$ and 1.01× for $n = 30$. This shows that as the function complexity increases, virtine start-up overheads become negligible, as expected. Here we can amortize start-up overheads with ~100$\mu$s of work.

We compare virtine start-up costs to the cost of crossing isolation boundaries in other published systems in Table 2. While the types of isolation these systems provide is slightly different, these numbers put the cost of the underlying mechanism into perspective. LwC and Enclosures switch between isolated contexts within the same kernel in a similar way to process-based isolation. SeCage and Hodor measure only the latency of the VMFUNC instruction without a VMEXIT event. Virtine latency is measured from userspace on the host, surrounding the KVM_RUN ioctl, thus incurring system call and ring-switch overheads.

### 6.2 Impact of Image Size

To evaluate the impact of virtines' execution environments on start-up costs, we performed an experiment that artificially increases image size, shown in Figure 12. This figure shows increasing virtine image size (up to 16MB) versus virtine execution latency for a minimal virtine that simply halts on startup. We synthetically increase image size by padding a minimal

virtine image with zeroes. With a 16MB image size, the start-up cost is 2.3ms. This amounts to roughly 6.8GB/s, which is in line with our measurement of the `memcpy` bandwidth on our *tinker* machine, 6.7GB/s. This shows the minimal cost a virtine will incur for start-up with a simple snapshotting strategy when the boot sequence is eliminated. Using a copy-on-write approach, as is done in SEUSS [21], we expect this cost could be reduced drastically.

These results reflect what others have seen for unikernel boot times. Unikernels tend to have a larger image size than what would be needed for a virtine execution environment, and thus incur longer start-up times. Kuenzer et al. report the shortest we have seen, at 10s to 100s of $\mu$s for Unikraft [42], while other unikernels (MirageOS [54], OSv [40], Rump [39], HermiTux [61], and Lupine [43]) take tens to hundreds of milliseconds to boot a trivial image. For example, we measured the no-op function evaluation time under OSv to be roughly 600 milliseconds on our testbed. A similar no-op function achieved roughly 12ms under MirageOS run with Solo5's HVT tender [12], which directly interfaces with KVM and uses hypercalls in a similar way to virtines.

### 6.3 Host Interaction Costs

As outlined in Section 2, virtines must interact with the client for all actions that are not fulfilled by the environment within the virtine. For example, a virtine must use hypercalls to read files or access shared state. Here we attempt to determine how frequent client interactions (via hypercalls) affect performance for an easily understood example. To do so, we use our C extension to annotate a connection handling function in a simple, single-threaded HTTP server that serves static content. Each connection that the server receives is passed to this function, which automatically provisions a virtine environment.

We measured both the latency and throughput of HTTP requests with and without virtines on *tinker*. The results are shown in Figure 13. Virtine performance is shown with and without snapshotting ("virtine" and "snapshot"). Requests are generated from `localhost` using a custom request generator (which always requests a single static file). Note that each virtine invocation here involves seven host interactions (hypercalls): (1) `read()` a request from host socket, (2) `stat()` requested file, (3) `open()` file, (4) `read()` from file, (5) `write()` response, (6) `close()` file, (7) `exit()`. Wasp handles these hypercalls by first validating arguments, and if they are allowed through, re-creates the calls on the host. For example, a validated `read()` will turn into a `read()` on the host filesystem. The exits generated by these hypercalls are doubly expensive due to the ring transitions necessitated by KVM. However, despite the cost of these host interactions, virtines with snapshots incur only a 12% decrease in throughput relative to the baseline. We expect that these costs would be reduced in a more realistic HTTP server, as more work unrelated to I/O would be involved. This effect has been observed by others employing connection sandboxing [27].

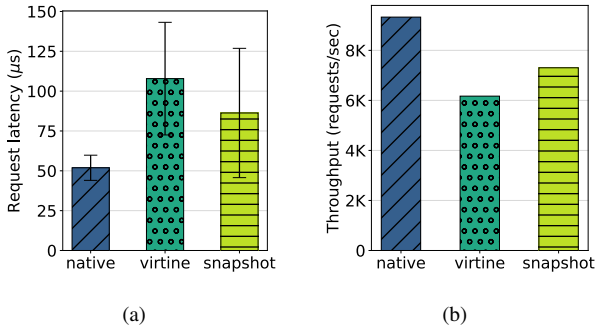


**Figure 13.** Mean response latency (a) and harmonic mean of throughput (b) for a simple HTTP server written in C, with each request handled natively and in a virtine (with and without snapshotting).

### 6.4 Integration with Library Code

To investigate the difficulty of incorporating virtines into libraries, and more significant codebases, we modified off-the-shelf OpenSSL.[4] OpenSSL is used as a library in many applications, such as the Apache web server, Lighttpd, and OpenVPN. We changed the library so that its 128-bit AES block cipher encryption is carried out in virtine context. We chose this function since it is a core component of many higher-level encryption features. While this would not be a good candidate for running in virtine context from a performance perspective, it gives us an idea of how difficult it is to use virtines to isolate a deeply buried, heavily optimized function in a large codebase.

Compiling OpenSSL using virtines was straightforward. From the developer's perspective, it simply involved annotating the block cipher function with the `virtine` keyword and integrating our custom clang/LLVM toolchain with the OpenSSL build environment (i.e., swapping the default compiler). The latter step was more work. In all, the change took roughly one hour for an experienced developer.

Though our main goal here was not to evaluate end-to-end performance, we did measure the performance impact of integrating virtines using OpenSSL's internal benchmarking tool. We ran the built-in speed benchmark[5] to measure the throughput of the block cipher using virtines (with our snapshotting optimization) compared to the baseline (native execution). Note that since the block cipher is being invoked many thousands of times per second, virtine creation overheads amplify the invocation cost significantly. In a realistic

[4] OpenSSL version 3.0.0 alpha7.

[5] `openssl speed -elapsed -evp aes-128-cbc`

scenario, the developer would likely include more functionality in virtine context, amortizing those overheads. That said, with our optimizations and a 16KB cipher block size, virtines only incur a 17× slowdown relative to native execution with snapshotting. The OpenSSL virtine image we use is roughly 21KB, which following Figure 12 will translate to 16$\mu$s for every virtine invocation. It follows, then, that virtine creation in this example is memory bound, since copying the snapshot comprises the dominant cost.

### 6.5 Virtines for Managed Languages

As described in our threat model (§3.2), virtines can provide isolation in environments where untrusted code executes. Examples of such environments are serverless platforms and databases UDFs. These environments often use high-level languages like JavaScript, Python, or Java to isolate the untrusted code. However, this isolation can still be compromised by bugs in the isolation logic.

Motivated by these environments, we investigate how a managed language can incorporate fine-grained isolation by running JavaScript functions in virtine context, and by exploring how virtine-specific optimizations can be used to reduce costs and improve latencies.

***Implementation.*** We chose the Duktape JavaScript engine for its portability, ease-of-use, and small memory footprint [10]. Our baseline implementation (no virtines) is configured to allocate a Duktape context, populate several native function bindings, execute a function that base64 encodes a buffer of data, and returns the encoding to the caller after tearing down (freeing) the JS engine. The virtine does the same thing, but uses the Wasp runtime library directly (no language extensions). This allows the engine to use only three hypercalls: `snapshot()`, `get_data()`, and `return_data()`. The snapshot hypercall instructs the runtime to take a snapshot after booting into long mode and allocating the Duktape context. `get_data()` asks the hypervisor to fill a buffer of memory with the data to be encoded, and once the virtine encodes the data, it calls `return_data()` and the virtine exits. By co-designing the hypervisor and the virtine, and by providing only a limited set of hypercalls, we limit the attack surface available to an adversary. For example, `snapshot` and `get_data` cannot be called more than once, meaning that if an attacker were to gain remote code execution capabilities, the only permitted hypercall would terminate the virtine.

***Benchmarking and evaluation.*** Figure 14 shows the results of our Duktape implementation. The virtine trial without snapshotting takes 125$\mu$s longer to execute than the baseline. We attribute this to several sources, including the required virtine provisioning and initialization overhead and the overhead to allocate and later free the Duktape context. By giving programmers direct control over more aspects of the execution environment, several optimizations can be made. For example, snapshotting can be used as shown in Figure 7 by

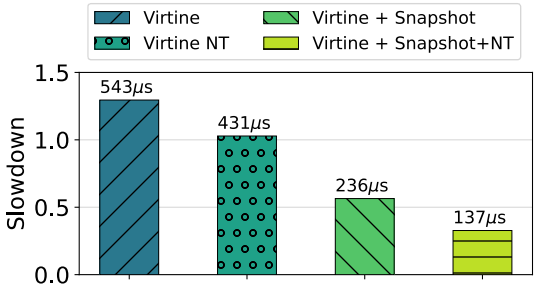


**Figure 14.** Slowdown of JavaScript virtines relative to native. The baseline latency is 419$\mu$s.

including the initialization of the JavaScript engine in the virtine's boot sequence. Doing so avoids many calls to `malloc` and other expensive functions while initializing. By taking advantage of snapshotting in the case of the "Virtine + Snapshot" measurements, virtines can enjoy a significant reduction in overhead–roughly 2×. Further, since all virtines are cleared and reset after execution, paying the cost of tearing down the JavaScript engine can be avoided. By applying both of these optimizations, the virtine can almost entirely avoid the cost of allocating and freeing the Duktape context by retaining it–something that cannot be done when executing in the client environment. Both of the trials, "Virtine NT" and "Virtine+Snapshot+NT" are designed to take advantage of this "**N**o **T**eardown" optimization in full. Note that the virtine is not executing code any *faster* than native, but it is able to provide a significant reduction in overhead by simply executing less code by applying optimizations. These optimizations cause the overall latency to drop to 137$\mu$s, which effectively constitutes the parsing and execution of the JavaScript code. Similar optimizations are applied in SEUSS [21], which uses the more complex V8 JavaScript engine, and thus avoids even more initialization overhead.

## 7 Discussion

In this section, we discuss how our results might translate to realistic scenarios and more complex applications, the limitations of our current approach, and other use cases we envision for virtines.

### 7.1 Implications

***Libraries.*** In Section 6.4, we demonstrated that it requires little effort to incorporate virtines into existing codebases that use sensitive or untrusted library functions. In our example we assumed access to the library's code (`libopenssl` in our case). While others make the same assumption [27], this is not an inherent limitation. The virtine runtime could apply a combination of link-time wrapping and binary rewriting

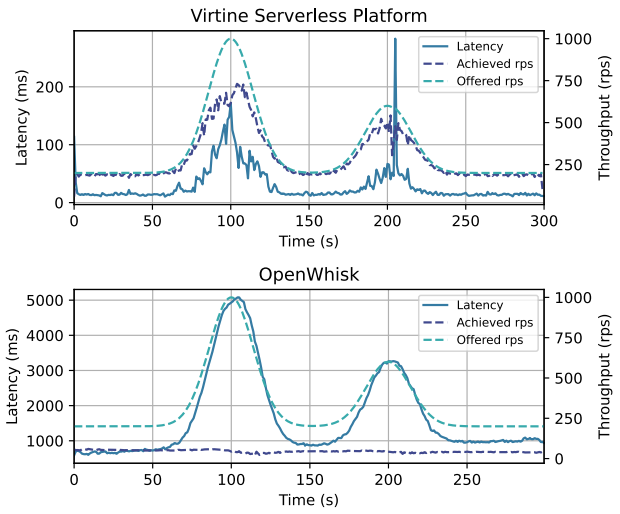


**Figure 15.** Serverless virtine performance compared to OpenWhisk's container-based platform.

to migrate library code automatically to run in virtine context. Others have applied such techniques for software fault isolation (SFI) [74], even in virtualized settings [31].

***Serverless Functions.*** While production serverless platforms and databases may use more complete JavaScript engines like V8, we can reason about how the results of our Duktape implementation would translate to these settings. Amazon Lambda, for example, constructs a container to achieve the desired level of isolation [73]. Here, the cost of creating a process and allocating a V8 context is considerable. If similar or better isolation can be achieved with a virtine, then the cost of creating the container can be eliminated.

To determine the feasibility of serverless virtines, we implemented a prototype serverless platform based on Apache's OpenWhisk framework [62] that integrates with our virtine Duktape engine. The current implementation only supports a single-node setup, and does not yet incorporate an API gateway, a load balancer, or authentication mechanisms. In this platform, which we call Vespid, users register JavaScript functions via a web application, which produces requests to our framework's main endpoint. These requests are handled by a concurrent server which runs each serverless function in a distinct virtine (rather than a container) by leveraging the Wasp runtime API. We measure the performance of invoking the same base64 JS encoding function used in Section 6.5 on our Vespid platform (which uses Duktape) and compare it to vanilla OpenWhisk (which uses V8 via Node.js). The results are shown in Figure 15. We measure end-to-end latency of both platforms, where client requests are generated on the same node as the server to minimize differences in the front-end implementations. We produce a series of concurrent function requests (from multiple clients) against both platforms using Locust [52], an off-the-shelf workload generator. This invocation pattern involves an initial ramp-up period that leads to two bursts, which then ramp down. Achieved throughput is shown on the dotted line. Vespid benefits from the lightweight virtine execution environment, Duktape's small image size, and Wasp's caching and snapshotting optimizations, leading to low-latency responses for a bursty load pattern. However, it is important to note that Vespid is a prototype that lacks many features offered by OpenWhisk, including the high-performance V8 engine. Note also that OpenWhisk's container engine does not employ optimizations such as container reuse and snapshotting seen in the recent literature like SOCK [60], SEUSS [21], Faasm [70], and Catalyzer [26], which all provide cold-start latencies less than 20ms. These results do show that a virtine-based serverless platform with competitive performance is feasible. Serverless functions that leverage external resources like S3 buckets can be facilitated with the appropriate virtine client support via hypercall handlers.

***User-defined Fucntions.*** A similar model could be used to more strongly isolate UDFs from one another in database systems. Postgres, for example, uses V8 mechanisms to isolate individual UDFs from one another [11], but they still execute in the same address space. Because virtine address spaces are disjoint, they could help with this limitation. Furthermore, virtines would allow functions in unsafe languages (e.g., C, C++) to be safely used for UDFs.

### 7.2 Limitations

***Workloads.*** While our current workloads represent components that could be used in real settings, we do not currently integrate with commodity serverless platforms or database engines. This integration is currently underway, with OpenWhisk and PostgreSQL, respectively.

Applications that rely on high-performance JavaScript will use a production engine like V8 or SpiderMonkey. Our evaluation uses Duktape, which lacks features like JIT compilation, but has the benefits of compiling into a small (~578KB) image and being easily portable. We envision that with sufficient runtime support—in particular, a port of the C++ standard library—a V8 virtine implementation is likely feasible. We believe our evaluation demonstrates the potential for incorporating virtines with high-level languages (HLLs).

***Language extensions.*** Currently, our C extension lacks the ability to take advantage of functionality located in a different LLVM module (C source file) than the one that contains the C function. Build systems used in C applications produce intermediate object files that are linked into the final executable. This means that virtines created using the C extension are restricted to functionality in the same compilation unit. Solutions to this problem typically involve modifying the build system to produce LLVM bitcode and using whole program

analysis to determine which functions are available to the virtine, and which are not.

Automatically generated virtines face an ABI challenge for argument passing. Because they do not share an address space with the host, argument marshalling is necessary. We leveraged LLVM to copy a compile-time generated structure containing the argument values into the virtine's address space at a known offset. Marshalling does incur an overhead that varies with the argument types and sizes, as is typical with "copy-restore" semantics in RPC systems [19]. This affects start-up latencies when launching virtines, as described in §6.4.

Virtines do not currently support nesting, but this is not an inherent limitation. Virtines that dynamically allocate memory are possible with an execution environment that provides heap allocation, but that memory is currently limited to the virtine context. We believe secure channels to communicate data between the virtine and host could be implemented with appropriate hypercalls and library/language support. The virtine compiler could identify and transform such allocation sites (e.g., malloc) using escape analysis.

***Snapshotting performance.*** Wasp's snapshotting mechanism currently uses `memcpy` to populate a virtine's memory image with the snapshot. This copying, as shown in Figure 12, constitutes a considerable cost for a large virtine image. We expect this cost to drop when using copy-on-write mechanisms to reset a virtine, as in SEUSS [21].

***KVM performance.*** As we found in Section 4.2, KVM has performance penalties due to its need to perform several ring transitions for each exit, and for VM start-up. Some of these costs are unavoidable because they maintain userspace control over the VM. However, a Type-I VMM like Palacios or Xen [16, 44] can mitigate some software latencies incurred by virtines.

***Security.*** Our threat model makes assumptions that may not hold in the real world. For example, a hardware bug in VT-x or a microarchitectural side channel vulnerability (e.g., Meltdown [47]) could feasibly be used to break our security guarantees.

### 7.3 Other use cases

In our examples, we used virtines to isolate certain annotated functions from the rest of the program. This use case is not the only possible one. Below, we outline several other potential use cases for virtines.

***Augmenting language runtimes.*** We believe that HLLs present an *incremental* path to using virtines, i.e., the language runtime might abstract away the use of virtines entirely, for example, to wrap function calls via the foreign function interface (Chisnall et al. employed special-purpose hardware for this purpose [22]). Virtines might also be used to apply security-in-depth to JIT compilers and dynamic binary translators. For example, bugs that lead to vulnerabilities in built-in functions or the JIT's slow-path handlers [66] can be mitigated by running them in virtine context (NoJITsu achieves this with Intel's Memory Protection Keys [64]). Polyglot environments like GraalVM [76] could more safely use native code by employing virtines.

***Distributed services.*** Because virtines implement an abstract machine model, are packaged with their runtime environment, and employ similar semantics to RPC [19], they allow for location transparency. Virtines could therefore be migrated to execute on remote machines just like containers, e.g., for code offload. This could allow for implementing distributed services with virtines, and for service migration based on high load scenarios, especially when RPCs are fast, as in the datacenter [38]. If virtines require host services or hardware not present in the local machine, they can be migrated to a machine that *does*.

## 8 Related Work

There is significant prior work on isolation of software components. However, the received wisdom is that when using hardware virtualization, creating a new isolated context for every isolation boundary crossing is too expensive. With virtines, we have shown that, with sufficient optimization, these overheads can be significantly reduced. Virtines enjoy several unique properties: they have an easy-to-use programming model, they implement an abstract machine model that allows for customization of the execution environment and the hypervisor, and because they create new contexts on every invocation, we can apply snapshotting to optimize start-up costs. We now summarize key differences with prior work.

***Isolation techniques.*** The closest work to virtines is Enclosures [27], which allow for programmer-guided isolation by splitting libraries into their own code, data, and configuration sections within the same binary. The security policy of Enclosures is defined in terms of *packages*, but with virtines, the security policy is defined and enforced at the level of individual functions. While, like Enclosures, virtines can be used to isolate library functions from their calling environment, they can *also* be used to selectively isolate functions from other users' virtines in a multi-tenant cloud environment.

Hodor [32] also provides library isolation, particularly for high-performance data-plane libraries. Gotee uses language-level isolation like virtines, but builds on SGX enclaves rather than hardware virtualization [28].

While TrustVisor [56] employs hardware virtualization to isolate application components (and assumes a strong adversary model), virtines enjoy a simpler programming model. SeCage uses static and dynamic analysis to automatically isolate software components guided by the secrets those components access [51]. Virtines give programmers more control

over isolated components. Glamdring also automatically partitions applications based on annotations [46], but uses SGX Enclaves which have more limited execution environments than virtines.

With Wedge [20], execution contexts (sthreads) are given minimal permissions to resources (including memory) using default deny semantics. However, virtines are more flexible in that they need not use the same host ABI and they do not require a modified host kernel. Dune is an example of an unconventional use of a virtual execution environment that provides high performance and direct access to hardware devices within a Linux system [18]. Unlike virtines, Dune's virtualization is at *process* granularity. Similarly, SMV isolates multi-threaded applications [33].

Several systems that support isolated execution leverage Intel's Memory Protection Keys [37] for memory safety [24, 35, 64, 68, 72]. For virtines, we chose not to use this mechanism since the number of protection domains (16) offered by the hardware was insufficient for multi-tenant scenarios (e.g., serverless). Even without this limitation, instructions that access the PKRU register would need to be validated/removed, e.g., with binary rewriting, as is done in ERIM [72]. We leave the exploration of MPK and similar fine-grained memory protection mechanisms for future work.

Lightweight-Contexts (LwCs) are isolated execution contexts *within* a process [48]. They share the same ABI as other contexts, but essentially act as isolated co-routines. Unlike LwCs, virtines can run an arbitrary software stack, and gain the strong isolation benefits of hardware virtualization. The Endokernel architecture [35] enables intra-process isolation with virtual privilege rings, but still maps domains to the process abstraction, rather than functions.

Nooks [71], LXD [58], and Nested Kernel [25] all implement isolation for kernel modules.

Software Fault Isolation [74] (SFI) enforces isolation by instrumenting applications with enforcement checks at boundary crossings, and thus does not leverage hardware support.

***Virtualization.*** Wasp is similar in architecture to other minimal hypervisors (implementing $\mu$VMs). Unlike Amazon's Firecracker [13] or Google's Cloud Hypervisor [30], we do not intend to boot a full Linux (or Windows) kernel, even with a simplified I/O device model. Wasp bears more similarity to ukvm [75] (especially the networking interface) and uhyve [45]. Unlike those systems, we designed Wasp to use a set of pre-packaged runtime environments. We intend Wasp to be used as a pluggable back-end (for applications, libraries, serverless platforms, or language runtimes) rather than as a stand-alone VMM.

***Serverless.*** Faasm [70] employs SFI for isolated, stateful serverless functions, partly based on the premise that hardware virtualization is simply too expensive. We show that this is not necessarily the case. Cloudflare uses V8 JavaScript "isolates" to reduce VM cold-start at the language level [23]. Others have developed optimized serverless systems based on the observation that a significant fraction of start-up costs can be attributed to language runtime and system software initialization, a task often duplicated across function invocations [21, 26, 60]. These systems achieve start-up latencies in the sub-millisecond range with aggressive caching of runtime state, which we also employ.

***Execution environments.*** Jitsu [53] allows Unikernels to be spawned on demand in response to network events, but does not allow programmers to invoke virtualized environments at the function call granularity. There is a rich history of combining language and OS research. Typified by MirageOS [54], writing kernel components in a high-level language gives the kernel developer more flexibility in moving away from legacy interfaces. It can also shift the burden of protection and isolation [34, 65].

## 9 Conclusions and Future Work

In this work, we explored the design and implementation of *virtines*—light-weight, isolated, virtualized functions, which can provide fine-grained execution without much of the overheads of traditional hardware virtualized execution environments. We probed the lower limits of hardware virtualization and presented Wasp, an embeddable hypervisor designed for virtines with microsecond start-up latency and limited slowdown. Wasp allows programs to easily create isolated contexts with tunable isolation policies. We introduced a compiler extension that allows developers to use virtines with simple code annotations, and explored how they can be used to ease virtine deployment. We demonstrated that integrating virtines with existing library code takes little effort. We modified an off-the-shelf JavaScript engine to use virtines, and explored how high-level languages can take advantage of them. In future work, we plan to explore other virtine applications in HLLs, for example in JIT engines. We also plan to investigate automatic virtine environment synthesis.

## Acknowledgments

We thank the anonymous reviewers and our shepherd, Martin Maas. We also thank the anonymous reviewers from OSDI '20 and EuroSys '21 for their encouraging and constructive feedback that helped improve the paper significantly. We thank Andrew Chien, Tyler Caraza-Harter, Boris Glavic, and Peter Dinda for enlightening discussions and suggestions. We also thank Andrew Neth, Cooper Van Kampen, and Griffin Dube for their help field testing the artifact scripts on a variety of machines. This work was made possible with support from the United States National Science Foundation (NSF) via grants CNS-1718252, CNS-1730689, REU-1757964, and CNS-1763612.

# References


[1] [n.d.]. *Chrome Sandbox - Linux Implementation Details*. Retrieved May 3, 2021 from https://chromium.googlesource.com/chromium/src/+/master/docs/linux/sandboxing.md

[2] 2009. CVE-2009-2555. Available from MITRE, CVE-ID CVE-2009-2555.. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2555

[3] 2009. CVE-2009-2935. Available from MITRE, CVE-ID CVE-2009-2935.. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2935

[4] 2009. CVE-2014-0160. Available from MITRE, CVE-ID CVE-2014-0160.. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160

[5] 2009. CVE-2018-18342. Available from MITRE, CVE-ID CVE-2018-18342.. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-18342

[6] 2009. CVE-2018-6056. Available from MITRE, CVE-ID CVE-2018-6056.. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-6056

[7] 2009. CVE-2021-3156. Available from MITRE, CVE-ID CVE-2021-3156.. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3156

[8] 2017. CVE-2009-2555. Available from NVD, CVE-ID CVE-2017-2505.. https://nvd.nist.gov/vuln/detail/CVE-2017-2505

[9] 2018. *Newlib*. Retrieved May 20, 2020 from https://sourceware.org/newlib/

[10] 2022. *Duktape Javascript Engine*. https://duktape.org/

[11] 2022. *Procedural Languages in PostgreSQL*. https://www.postgresql.org/docs/13/external-pl.html

[12] 2022. *Solo5*. https://github.com/Solo5/solo5

[13] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proceedings of the* $17^{th}$ *USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*. USENIX Association, Santa Clara, CA, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[14] Kalev Alpernas, Cormac Flanagan, Sadjad Fouladi, Leonid Ryzhyk, Mooly Sagiv, Thomas Schmitz, and Keith Winstein. 2018. Secure Serverless Computing Using Dynamic Information Flow Control. *Proceedings of the ACM on Programming Languages* 2, OOPSLA, Article 118 (Oct. 2018), 26 pages. https://doi.org/10.1145/3276488

[15] AMD Corporation 2016. *AMD64 Architecture Programmer's Manual Volume 2: Systems Programming*. AMD Corporation.

[16] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. In *Proceedings of the* $19^{th}$ *ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 164–177. https://doi.org/10.1145/945445.945462

[17] Andrew Baumann, Dongyoon Lee, Pedro Fonseca, Lisa Glendenning, Jacob R. Lorch, Barry Bond, Reuben Olinsky, and Galen C. Hunt. 2013. Composing OS Extensions Safely and Efficiently with Bascule. In *Proceedings of the* $8^{th}$ *ACM European Conference on Computer Systems* (Prague, Czech Republic) *(EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 239–252. https://doi.org/10.1145/2465351.2465375

[18] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of the* $10^{th}$ *USENIX Conference on Operating Systems Design and Implementation (OSDI '12)*. 335–348.

[19] Andrew D. Birrell and Bruce Jay Nelson. 1983. Implementing Remote Procedure Calls. In *Proceedings of the* $9^{th}$ *ACM Symposium on Operating Systems Principles (SOSP '83)*. https://doi.org/10.1145/800217.806609

[20] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the* $5^{th}$ *USENIX Symposium on Networked Systems Design and Implementation* (San Francisco, California) *(NSDI '08)*. USENIX Association, 309–322. https://www.usenix.org/conference/nsdi-08/wedge-splitting-applications-reduced-privilege-compartments

[21] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: Skip Redundant Paths to Make Serverless Fast. In *Proceedings of the* $15^{th}$ *European Conference on Computer Systems (EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 32, 15 pages. https://doi.org/10.1145/3342195.3392698

[22] David Chisnall, Brooks Davis, Khilan Gudka, David Brazdil, Alexandre Joannou, Jonathan Woodruff, A. Theodore Markettos, J. Edward Maste, Robert Norton, Stacey Son, Michael Roe, Simon W. Moore, Peter G. Neumann, Ben Laurie, and Robert N.M. Watson. 2017. CHERI JNI: Sinking the Java Security Model into the C. In *Proceedings of the* $22^{nd}$ *International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 569–583. https://doi.org/10.1145/3037697.3037725

[23] cloudflare [n.d.]. How Workers Works. https://developers.cloudflare.com/workers/learning/how-workers-works. Accessed 2021-05-01.

[24] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. 2020. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In *Proceedings of the* $29^{th}$ *USENIX Security Symposium (USENIX Security '20)*. USENIX Association, USA, 1409–1426.

[25] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. 2015. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. In *Proceedings of the* $20^{th}$ *International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 191—206. https://doi.org/10.1145/2694344.2694386

[26] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting. In *Proceedings of the* $25^{th}$ *International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 467–481. https://doi.org/10.1145/3373376.3378512

[27] Adrien Ghosn, Marios Kogias, Mathias Payer, James R. Larus, and Edouard Bugnion. 2021. Enclosure: Language-Based Restriction of Untrusted Libraries. In *Proceedings of the* $26^{th}$ *ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 255–267. https://doi.org/10.1145/3445814.3446728

[28] Adrien Ghosn, James R. Larus, and Edouard Bugnion. 2019. Secured Routines: Language-based Construction of Trusted Execution Environments. In *Proceedings of the USENIX Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, 571–586. http://www.usenix.org/conference/atc19/presentation/ghosn

[29] Michael Godfrey, Tobias Mayr, Praveen Seshadri, and Thorsten von Eicken. 1998. Secure and Portable Database Extensibility. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data* (Seattle, Washington, USA) *(SIGMOD '98)*. Association for Computing Machinery, New York, NY, USA, 390—401. https://doi.org/10.1145/276304.276339

[30] Google, Inc. 2021. *Google Cloud Hypervisor*. Retrieved January 1, 2020 from https://github.com/cloud-hypervisor/cloud-hypervisor

[31] Kyle C. Hale and Peter A. Dinda. 2014. Guarded Modules: Adaptively Extending the VMM's Privilege Into the Guest. In *Proceedings of the* $11^{th}$ *International Conference on Autonomic Computing* (Philadelphia, PA) *(ICAC '14)*. USENIX Association, 85–96. https://www.usenix.org/conference/icac14/technical-sessions/presentation/hale

[32] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '19)*. USENIX Association, Renton, WA, 489–504. https://www.usenix.org/conference/atc19/presentation/hedayati-hodor

[33] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. 2016. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 393––405. https://doi.org/10.1145/2976749.2978327

[34] Galen C. Hunt and James R. Larus. 2007. Singularity: Rethinking the Software Stack. *SIGOPS Operating Systems Review* 41, 2 (April 2007), 37–49. https://doi.org/10.1145/1243418.1243424

[35] Bumjin Im, Fangfei Yang, Chia-Che Tsai, Michael LeMay, Anjo Vahldiek-Oberwagner, and Nathan Dautenhahn. 2021. The Endokernel: Fast, Secure, and Programmable Subprocess Virtualization. arXiv:2108.03705 [cs.CR]

[36] Intel Corporation 2017. *Intel® Software Guard Extensions SDK for Linux OS*. Intel Corporation.

[37] Intel Corporation 2021. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B & 3C): System Programming Guide*. Intel Corporation.

[38] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation* (Boston, MA) *(NSDI '19)*. USENIX Association, 1–16. https://www.usenix.org/conference/nsdi19/presentation/kalia

[39] Antti Kantee. 2012. *The Design and Implementation of the Anykernel and Rump Kernels*. Ph.D. Dissertation. Aalto University, Helsinki, Finland.

[40] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. 2014. OSv—Optimizing the Operating System for Virtual Machines. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC '14)*.

[41] Ricardo Koller and Dan Williams. 2017. Will Serverless End the Dominance of Linux in the Cloud?. In *Proceedings of the* $16^{th}$ *Workshop on Hot Topics in Operating Systems* (Whistler, BC, Canada) *(HotOS XVI)*. Association for Computing Machinery, New York, NY, USA, 169–173. https://doi.org/10.1145/3102980.3103008

[42] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. 2021. Unikraft: Fast, Specialized Unikernels the Easy Way. In *Proceedings of the* $16^{th}$ *European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 376–394. https://doi.org/10.1145/3447786.3456248

[43] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. 2020. A Linux in Unikernel Clothing. In *Proceedings of the* $15^{th}$ *European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3342195.3387526

[44] John Lange, Kevin Pedretti, Trammell Hudson, Peter Dinda, Zheng Cui, Lei Xia, Patrick Bridges, Andy Gocke, Steven Jaconette, Mike Levenhagen, and Ron Brightwell. 2010. Palacios and Kitten: New High Performance Operating Systems for Scalable Virtualized and Native Supercomputing. In *Proceedings of the* $24^{th}$ *IEEE International Parallel and Distributed Processing Symposium (IPDPS'10)*.

[45] Stefan Lankes, Simon Pickartz, and Jens Brietbart. 2017. A Low Noise Unikernel for Extreme-Scale Systems. In *Proceedings of the* $30^{th}$ *International Conference on Architecture of Computing Systems (ARCS '17)*.

[46] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter Pietzuch. 2017. Glamdring: Automatic Application Partitioning for Intel SGX. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '17)*. USENIX Association, Santa Clara, CA, 285–298. https://www.usenix.org/conference/atc17/technical-sessions/presentation/lind

[47] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of the* $27^{th}$ *USENIX Security Symposium (USENIX Security '18)*. USENIX Association, Baltimore, MD, 973–990. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[48] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *Proceedings of the* $12^{th}$ *USENIX Symposium on Operating Systems Design and Implementation* (Savannah, GA) *(OSDI '16)*. USENIX Association, 49–64. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/litton

[49] Conghao Liu and Kyle C. Hale. 2019. Towards a Practical Ecosystem of Specialized OS Kernels. In *Proceedings of the International Workshop on Runtime and Operating Systems for Supercomputers* (Phoenix, AZ, USA) *(ROSS '19)*. Association for Computing Machinery, New York, NY, USA, 3–9. https://doi.org/10.1145/3322789.3328742

[50] Lei Liu, Xinwen Zhang, Vuclip Inc, Guanhua Yan, and Songqing Chen. 2012. Chrome extensions: Threat analysis and countermeasures. In *In 19th Network and Distributed System Security Symposium (NDSS '12*.

[51] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting Memory Disclosure with Efficient Hypervisor-Enforced Intra-Domain Isolation. In *Proceedings of the* $22^{nd}$ *ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 1607—1619. https://doi.org/10.1145/2810103.2813690

[52] locust [n.d.]. *Locust: An open source load testing tool*. Retrieved February 20, 2022 from https://locust.io/

[53] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian Leslie. 2015. Jitsu: Just-In-Time Summoning of Unikernels. In *Proceedings of the* $12^{th}$ *USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*. Oakland, CA, 559–573. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/madhavapeddy

[54] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the* $18^{th}$ *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. 461–472.

[55] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) Than Your Container. In *Proceedings of the* $26^{th}$ *Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 218–233. https://doi.org/10.1145/3132747.

3132763

[56] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. 2010. TrustVisor: Efficient TCB Reduction and Attestation. In *Proceedings of the IEEE Symposium on Security and Privacy* (Oakland, CA, USA) *(S&P '10)*. IEEE, 143–158. https://doi.org/10.1109/SP.2010.17

[57] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. 2008. Flicker: An Execution Infrastructure for Tcb Minimization. In *Proceedings of the* $3^{rd}$ *ACM SIGOPS European Conference on Computer Systems* (Glasgow, Scotland UK) *(EuroSys '08)*. Association for Computing Machinery, New York, NY, USA, 315–328. https://doi.org/10.1145/1352592.1352625

[58] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. 2019. LXDs: Towards Isolation of Kernel Subsystems. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '19)*. USENIX Association, Renton, WA, 269–284. https://www.usenix.org/conference/atc19/presentation/narayanan

[59] NVIDIA Corporation 2020. *CUDA C++ Programming Guide—Version 11.1.0*. NVIDIA Corporation. Accessed: 2020-10-01.

[60] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *Proceedings of the USENIX Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, 57—-69. https://www.usenix.org/conference/atc18/presentation/oakes

[61] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. 2019. A Binary-Compatible Unikernel. In *Proceedings of the* $15^{th}$ *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Providence, RI, USA) *(VEE '19)*. Association for Computing Machinery, New York, NY, USA, 59–73. https://doi.org/10.1145/3313808.3313817

[62] openwhisk [n.d.]. Apache OpenWhisk. https://openwhisk.apache.org/. Accessed 2021-02-19.

[63] oracleudf [n.d.]. Oracle Database SQL Reference: User-Defined Functions. https://docs.oracle.com/cd/B19306_01/server.102/b14200/functions231.htm. Accessed 2021-05-01.

[64] Taemin Park, Karel Dhondt, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2020. NoJITsu: Locking Down JavaScript Engines. In *Proceedings of the Network and Distributed System Security Symposium* (San Diego, CA, USA) *(NDSS '20)*.

[65] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. 2011. Rethinking the Library OS from the Top Down. In *Proceedings of the* $16^{th}$ *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*. 291–304.

[66] Chris Rohlf and Yan Ivnitskiy. 2011. Attacking clientside JIT compilers. *Black Hat USA* (2011).

[67] Rusty Russell. 2008. Virtio: Towards a de-Facto Standard for Virtual I/O Devices. *SIGOPS Operating Systems Review* 42, 5 (July 2008), 95–103. https://doi.org/10.1145/1400097.1400108

[68] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. 2022. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. In *Proceedings of the* $31^{st}$ *USENIX Security Symposium* (Boston, MA, USA) *(USENIX Security '22)*. USENIX Association. https://www.usenix.org/conference/usenixsecurity22/presentation/schrammel

[69] sgx 2020. Intel® Software Guard Extensions. https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html. Accessed: 2020-08-06.

[70] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, 419–433. https://www.usenix.org/conference/atc20/presentation/shillaker

[71] Michael M. Swift, Steven Martin, Henry M. Levy, and Susan J. Eggers. 2002. Nooks: An Architecture for Reliable Device Drivers. In *Proceedings of the* $10^{th}$ *Workshop on ACM SIGOPS European Workshop* (Saint-Emilion, France) *(EW '10)*. Association for Computing Machinery, New York, NY, USA, 102–107. https://doi.org/10.1145/1133373.1133393

[72] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *Proceedings of the* $28^{th}$ *USENIX Security Symposium (USENIX Security '19)*. USENIX Association, Santa Clara, CA, 1221–1238. https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner

[73] Tim Wagner. 2014. *Understanding Container Reuse in AWS Lambda*. Retrieved May 26, 2020 from https://aws.amazon.com/de/blogs/compute/container-reuse-in-lambda/

[74] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-Based Fault Isolation. In *Proceedings of the* $14^{th}$ *ACM Symposium on Operating Systems Principles* (Asheville, North Carolina, USA) *(SOSP '93)*. Association for Computing Machinery, New York, NY, USA, 203–216. https://doi.org/10.1145/168619.168635

[75] Dan Williams and Ricardo Koller. 2016. Unikernel Monitors: Extending Minimalism Outside of the Box. In *Proceedings of the* $8^{th}$ *USENIX Workshop on Hot Topics in Cloud Computing* (Denver, CO) *(HotCloud '16)*. USENIX Association, USA, 71–76. https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/williams

[76] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581

# A Artifact Appendix

## A.1 Abstract

This artifact comprises our Wasp $\mu$hypervisor, our Clang compiler extensions, our LLVM pass, the experimental code, scripts, and data used for the paper, as well as benchmarks and example code. Users should be able to re-create our experiments and compare results on a broad range of hardware. In our Github repo[6] we have detailed instructions on how to run Wasp, and compile code with virtine support.

## A.2 Description & Requirements

**A.2.1 How to access.** All code for virtines and Wasp is organized under the Virtines Github organization. This can be accessed at https://github.com/virtines. The primary repository that will be used for this Artifact is Wasp, which is at https://github.com/virtines/wasp. The DOI for our artifact is 10.5281/zenodo.6350453.

**A.2.2 Hardware dependencies.** Wasp currently supports x64 hardware, including AMD and Intel. Hardware virtualization extensions are a requirement (SVM on AMD, VT-x on Intel), so hardware from the last decade or so should work fine.

**A.2.3 Software dependencies.** The primary dependence is on a hosted hypervisor framework (namely, kvm on Linux). While early versions of Wasp ran on Hyper-V, the port of our compiler extensions and benchmarks is not yet complete, so for this artifact we focus on Linux. We also rely on the following packages and libraries:

- netwide assembler (`nasm`)
- `curl` development libraries (`libcurl-dev` on Debian-based distros)
- Clang C compiler (version 10 or newer)
- LLVM and development headers (`llvm` and `llvm-dev`)
- `cmake`
- Virtual environments for Python3 (`python3.8-venv`)

It is possible to run Wasp in a nested virtualization environment (as long as nested virtualization with kvm is enabled), but we recommend running on a bare-metal machine to get reasonable performance. More details on the build prerequisites can be found in the instructions in the repo (`README.md`).

**A.2.4 Benchmarks.** All relevant benchmarks are included in the repo. Experimental results are reproduced with these benchmarks as described below.

## A.3 Set-up

We recommend evaluators follow the guidance in our repo (`README.md`). To build Wasp, see the sections titled "Environment Setup," "Prerequisites," and "Building and Installing." As described there, we recommend an environment like Cloudlab or Chameleon Cloud. We provide a Cloudlab profile and instructions for Chameleon in our README document.

[6] https://github.com/virtines/wasp

## A.4 Evaluation workflow

Once Wasp is set up, reproduction of experimental results is a mostly automated process, though evaluators are free to focus on individual experiments. Before that, to ensure that Wasp is functional once built, users can run `make smoketest` to ensure everything is working properly. See our README ("Running Virtine Tests") for more detail. Once Wasp is functional, experimental results can be easily generated.

### A.4.1 Major Claims.

- *(C1): The core components of virtual context creation comprise only a few tens of thousands of cycles. We show this in experiment (E1) described in Section 4.2, whose results are shown in Table 1.*
- *(C2): The latency to run a function in different processor modes can vary (e.g., 16-bit mode is cheaper on some microarchitectures), presenting an opportunity for optimization when building virtual contexts. We show this in experiment (E2) in Section 4.2 of the paper, whose results are depicted in Figure 3.*
- *(C3): A runtime system that boots a basic server in a minimal execution environment can achieve response times <1ms, even without optimizations. We show this in experiment (E3) in Section 4.2 of the paper, whose results are shown in Figure 4.*
- *(C4): Virtual context creation latencies with Wasp approach the hardware limit of the vmrun/vmcall instruction by employing optimizations. We show this in experiment (E4) in Section 5.2 of the paper (Figure 8). Note that we see Figure 2 as a subset of the results in Figure 8, so we elide it in the artifact.*
- *(C5): Virtine creation overheads can be amortized with roughly 100µs of work. In finer-grained scenarios, snapshotting can reduce overheads significantly, pushing the amortization point down by about 10×. We show this in experiment (E5) in Section 6.1 of the paper (Figure 11).*
- *(C6): Once virtine image size reaches around 2MB, start-up latency becomes bottlenecked by memory bandwidth. We show this in experiment (E6) in Section 6.2 of the paper (Figure 12).*
- *(C7): An HTTP server using our virtine compiler extensions experiences a less than 20% drop in throughput relative to a native environment. We show this in experiment (E7) in Section 6.3 of the paper (Figure 13).*
- *(C8): Virtines can be integrated with an off-the-shelf Javascript engine, with acceptable ($< 1.5\times$) slow-downs (~2×). Snapshotting improves performance when environment setup in the virtual context is non-trival. We*

*show this in experiment (E8) in Section 6.5 of the paper (Figure 14).*

**A.4.2 Experiments.** To re-run all experiments, you can simply run `make artifacts.tar` as described in our README. This will take roughly five minutes to run to completion and generate data and plots, which can then be found in `artifacts.tar`. Once the results are generated, they can be compared with the data from the paper (which can be also be found in `data_example/gold/`). We have also provided data we have generated on many other machines. These can also be found in `data_example/*`; the machine and software environment descriptions can be found in `data_example/README.md`.

We outline individual experiments below.

*Experiment (E1): [Boot Breakdown] [1 sec.]: This experiment evaluates the various components that comprise booting a virtual context. Use this to evaluate claim (C1).*

*[How to]* Run `make figure1_data`.

*[Results]* The results will appear in `data/figure1_data.csv`. You should see that the total average cycle counts less than ~100K cycles (ignoring the first run). The transition to protected mode (`prot`) and the identity mapping (`id map`) will be the most expensive components of the boot process.

*Experiment (E2): [Mode latency] [5 sec.]: This experiment evaluates the time to run a recursive implementation of fib(20) in 16-bit mode, 32-bit (protected) mode, and 64-bit (long) mode. Use this to evaluate claim (C2).*

*[How to]* Run `make fig3.pdf`.

*[Results]* You can see the results in `fig3.pdf`. In most machines the time to run the function will vary with processor modes, but on some there is little difference. The point here is that there is room for a virtine compiler to leverage hardware knowledge to optimize code generated for virtine context.

*Experiment (E3): [Echo server] [5 sec.]: This experiment shows that with a minimal virtual exection context, an HTTP server can achieve response times <1ms. Use this to evaluate claim (C3).*

*[How to]* Run `make fig4.pdf`.

*[Results]* You can see the results in `fig4.pdf`. You should see that the time to get an HTTP response should be somewhere between 100K and 500K cycles.

*Experiment (E4): [Context creation] [5 sec.]: This experiment shows that Wasp can achieve start-up latencies close to the hardware limit (the vmrun/vmcall instruction on x86 hardware). Use this to evaluate claim (C4).*

*[How to]* Run `make fig8.pdf`.

*[Results]* You can see the results in `fig8.pdf`. You should see that the "Wasp+C" and "Wasp+CA" bars appear relatively close to the vmrun bar and outperform pthreads.

*Experiment (E5): [Virtine overheads] [30 sec.]: This experiment demonstrate how much computation is necessary to amortize virtine creation overheads. Use this to evaluate claim (C5).*

*[How to]* Run `make fig11.pdf`.

*[Results]* You can see the results in `fig11.pdf`. You should see the bars even out in the 100$\mu$s range, indicating that 100$\mu$s of computation are necessary to amortize creation overheads. Snapshotting should significantly improve call latency for smaller image sizes.

*Experiment (E6): [Image size impact] [3 sec.]: This experiment demonstrates the impact of image size on virtine creation latency. Use this to evaluate claim (C6).*

*[How to]* Run `make fig12.pdf`.

*[Results]* You can see the results in `fig12.pdf`. You should see a knee in the curve somewhere around 1-2MB. This is where virtine creation is becoming memory bandwidth bound. Where exactly the knee occurs depends on the memory copy bandwidth of the machine.

*Experiment (E7): [HTTP server] [1 min.]: This shows the latency and throughput of serving HTTP requests in a virtine (with and without snapshotting optimization) vs. native execution. Use this to evaluate claim (C7).*

*[How to]* Run `make fig13_lat.pdf` then `make fig13_tput.pdf`.

*[Results]* You can see the results in `fig13_lat.pdf` and `fig13_tput.pdf`. Expect to see a little more than 2× increase in latency and 2× drop in throughput relative to native. Snapshotting may actually reduce performance on this experiment on machines with limited memory bandwidth. Most of the performance drop is caused by hypercall interactions.

*Experiment (E8): [Javascript virtines] [3 sec.]: This shows the slowdown of launching Javascript virtines with various optimizations. Use this to evaluate claim (C8).*

*[How to]* Run `make fig14.pdf`.

*[Results]* You can see the results in `fig14.pdf`. You should see that snapshotting has a real effect given the amount of runtime initialization that takes place in the Duktape JavaScript engine. The slowdown for virtines without optimizations (the leftmost bar) should be in the 1.5–2× range.

### A.5 Notes on Reusability

If you would like to explore using embedded Wasp with your programs, we provide additional guidance in the "Embedding Wasp" section of our repo's README. Developers can interact with the runtime library directly using our API or indirectly using our compiler extensions.