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

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

Abstract

An important class of 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 uses extensions to the C language. We present a detailed analysis of the overheads involved with 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.

ACM Reference Format:
Nicholas C. Wanninger, Joshua J. Bowden, and Kyle C. Hale. 2022. Isolating Functions at the Hardware Limit with Virtines. In *EuroSys '22: ACM European Conference on Computer Systems, April 5–8, 2022, Rennes, France.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [15, 18, 20, 23–25, 37, 41–43, 52, 53]. 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 [21, 46], 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, the challenges[1] that arise from using the container abstraction for individual function execution—a usage model containers were not designed for—are formidable [16].

Guided by these examples, we introduce *virtines*, a new abstraction *designed* for isolating execution at function call granularity using hardware virtualization. Data touched by virtines is automatically encapsulated in their 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 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 both on 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.

[1] Particularly cold-start latency.

- We present a prototype embeddable hypervisor framework, Wasp, that implements the virtine abstraction. Wasp runs as a Type-II micro-hypervisor both on 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-incorporated versions incur acceptable slow-downs 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 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.[2] As with most code written to execute in a different environment from the host, (e.g., CUDA [44] or SGX enclaves [50]) 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 be passed arguments and can produce return values like any normal function.

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, and require the hypervisor interpose on all such requests. While the hypervisor provides the interposition mechanism, the virtine client has the option to implement a hypercall

[2]For example, like *goroutines*: https://gobyexample.com/goroutines

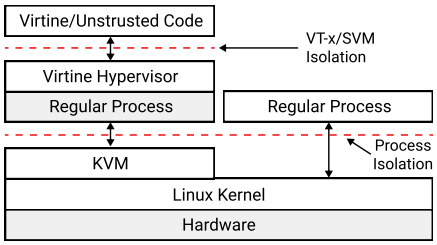

Figure 1. How virtines fit into the software stack.

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 of where the "cut" in the call graph is made by the programmer, but making this choice automatically in the compiler is possible [32, 37]. As a result of being only a subset of the call graph, virtine images are typically quite small (~16KB), and are statically compiled binaries containing all required software. Shared libraries violate requirements outlined in Section 3.1.

While the runtime environment *within* a virtine can vary, we expect they will almost always run in a limited, kernel-mode, environment. This may mean no scheduler, virtual 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 as shown in Dune [13]. 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 that these could be selected from a larger set given a rich virtine ecosystem. They could also be synthesized automatically.

3 Design

In this section we describe our isolation and safety objectives in developing the virtine abstraction, then we 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. Virtines must not have their private state be affected by another virtine running

untrusted or adversarial code. Thus, data secrecy must be maintained between virtines.

Virtine isolation. Virtines may not interact with any data or services outside of their own address space or what is permitted by the virtine client's policies, and host data secrecy must also be maintained.

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, 36]. Unfortunately, as with most software, bugs have allowed attackers to 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 do not assume a *strong* adversarial model; i.e., an attacker does *not* have access to host memory or execution state (including the hypervisor, client process, and host kernel). The hypervisor runtime and hypercall handlers are assumed to be implemented correctly. These handlers must take care to assume that inputs have *not* been properly sanitized, and may have even been 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 client memory. Additionally, we assume that microarchitectural side-channel mitigations are sufficient, and do not pose a threat.

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

All Linux and KVM experiments were run on *tinker*, an AMD EPYC 7281 (Naples; 16 cores; 2.69 GHz) machine wtih 32 GB DDR4 running stock Linux kernel version 5.9.12. We disabled hyperthreading, turbo boost, and DVFS to mitigate measurement noise. 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 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

Here 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.

Lower bounds. In the measurement of "vmrun" we are observing the lowest latency we could achieve to begin execution in a virtual context using KVM in Linux 5.9. This

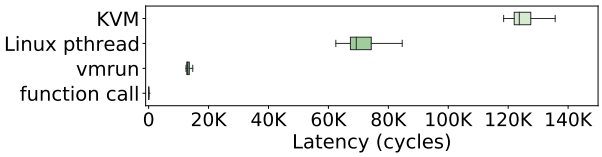

Figure 2. Lower bounds on execution context creation in cycles (measured with `rdtsc`).

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.

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.

Regardless of the shortcomings of the KVM interface, these measurements tell us that while a virtual function call 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 [16, 29, 40]. 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 [10]. Caching pre-booted environments can mitigate this overhead, as we describe in §5.2.

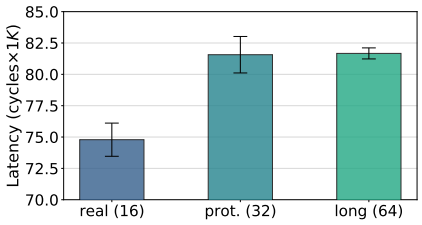

Figure 3. Latency to run a function in the three classic operating modes on x86. Note the use of a false origin to highlight relative differences.

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 number of 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 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 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 [11, 27]. 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), run $fib(20)$, and exit back to the host. These measurements

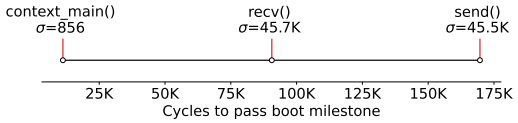

Figure 4. Latency for echo server startup milestones in protected mode (no paging).

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 here 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 potentially 10K cycles are on the table for savings. We plan to explore ways to determine whether a snippet of code can be run in real-mode and target it automatically to take advantage of this benefit in future work.

Booting up for useful code. We have seen that a minimal long-mode boot sequence costs less than 30K cycles (~12 μs), 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 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.

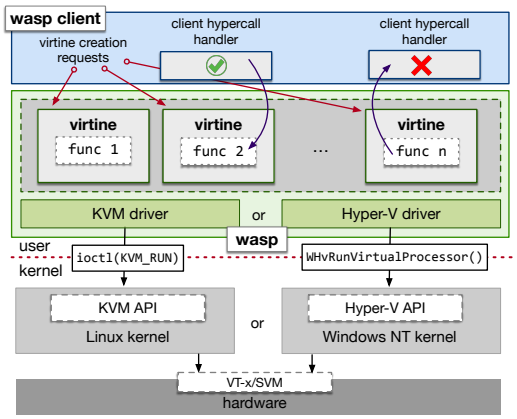

Figure 5. High-level overview of Wasp.

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 μs) *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 above the points.

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 virtines 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 [10], Unikernel monitors [56], and uhyve [31],

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 virtual devices, but are instead designed to provide hypervisor services with as few exits as possible. For example, rather than performing file I/O by ultimately interacting with a virtio device [49] and parsing filesystem structures, a virtine could use a hypercall that mirrors the `read` POSIX system call. Hypercalls are implemented by a co-designed handler that is invoked on each exit triggered by a hypercall. Wasp provides the mechanism to create virtines, while the client must provide the security policies through handlers. These handlers could simply run a series of checks and pass through certain host system calls while filtering others out. Wasp provides several default handlers that a client can use; these are also used by our language extensions (§5.3). A runtime could also fully isolate a virtine from the host by virtualizing a filesystem in memory, or by eliding filesystem access–or a hypercall interface–entirely. Note that 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 hypercall handlers, which are implemented (or selected) by the virtine client.

5.2 Wasp Caching and Snapshotting

Caching. To push virtine start-up latencies down, Wasp supports a pool of cached, uninitialized, virtines (shells) that can be reused. As depicted in Figure 6, the system receives a request externally (A), which will drive virtine creation. This external stimulus may come from network traffic, direct

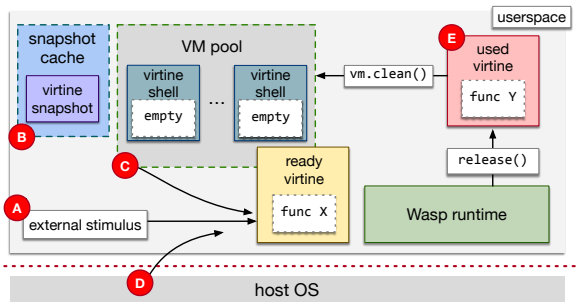

Figure 6. Image snapshotting and virtine reuse with a pooled design.

function execution requests from the application, or even database 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, (C) or (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 bound, 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 is meant to show that the caching mechanism brings the cost of provisioning a virtine shell to within 4% of a bare `vmrun`.

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 [16, 19, 45]. In a similar fashion, Wasp supports snapshotting by allowing a virtine to leverage

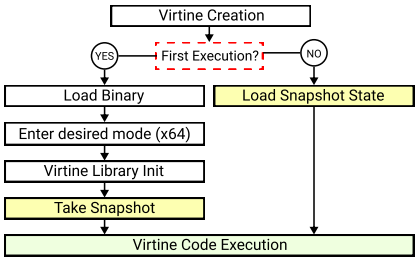

Figure 7. Virtine codepath on first execution and subsequent executions after snapshotting.

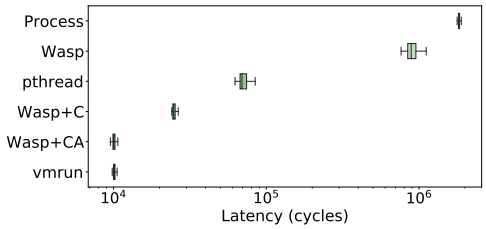

Figure 8. Creation latencies for execution contexts, including Wasp virtines. "Wasp+C" does not include snapshot benefits, as this measures the minimum execution latency. Note the log scale on the horizontal axis.

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 that detects C functions annotated with the `virtine` keyword, runs middle-end analysis at the IR level,

```
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.

and automatically generates code that invokes a pre-compiled virtine 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. Note that recursive calls to the function within the virtine do not result in nested virtine creations; in this case the system acts as if the virtine keyword was not included.

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, virtines do not intrinsically 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. The use of the `virtine` keyword denies all hypercalls, 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 bitmask 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

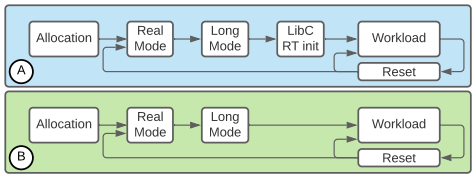

Figure 10. Execution environments available to a virtine.

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 [35].

6 Evaluation

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

- 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

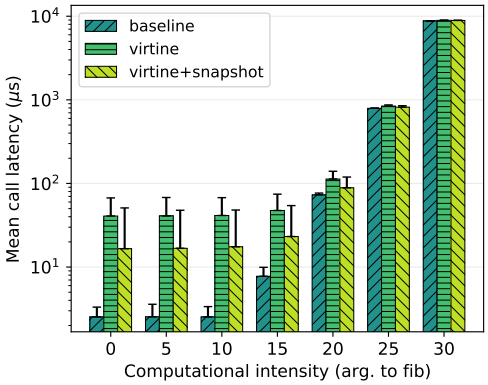

Figure 11. Latency of virtines as computational intensity increases. Note the log scale on the vertical axis. All run on *tinker*.

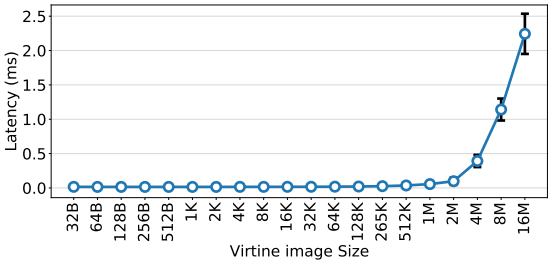

Figure 12. Impact of image size on start-up latency.

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.

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 ~100μs of work.

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 here 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 *tinker*, 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 [16], we expect this cost could be reduced drastically.

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 HTTP server. 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 [20].

6.4 Integration with Library Code

To investigate the difficulty of incorporating virtines into libraries, and more significant codebases generally, we modified off-the-shelf OpenSSL.[4] OpenSSL is used as a library in many applications, for example, the Apache webserver, 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

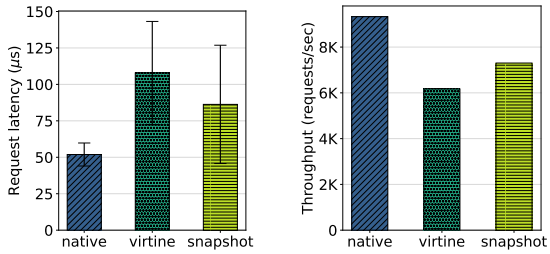

Figure 13. Mean response latency (a) and harmonic mean of throughput (b) for a simple HTTP server written in C, with each request handled natively and in a virtine (with and without snapshotting).

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.

Getting OpenSSL building 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. Because our current virtine compiler prototype lacks the ability to infer pointer sizes or handle variable-sized arguments, we did have to make modifications to split blocks into fixed-size regions.

Though our main goal with this exercise 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 simple 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 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 simple snapshotting. The OpenSSL virtine image we use is roughly 21KB, which following Figure 12 will translate to 16μs for every virtine invocation. It follows, then, that virtine creation in this example is memory bound, since copying the snapshot comprises the dominant cost.

[4] OpenSSL version 3.0.0 alpha7.

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

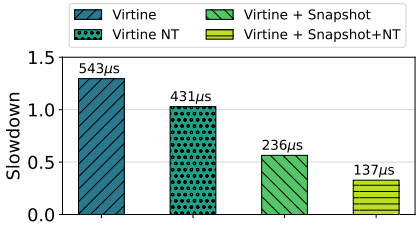

Figure 14. Slowdown of JavaScript virtines relative to native. The baseline latency is 419μs.

6.5 Virtines for Managed Languages

As described in our threat model (§3.2), virtines can excel at providing 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, but this isolation can still be compromised by bugs in the isolation logic.

Motivated by these environments, we investigate how a managed language can be isolated with virtines 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[6] for its portability, ease-of-use, and small memory footprint. 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 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 125μs 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 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 "leaking" 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 137μs, which effectively constitutes the parsing and execution of the JavaScript code. Similar optimizations are applied in SEUSS [16], which uses the more complex V8 JavaScript engine, and thus avoids even more initialization overhead.

[6] https://duktape.org/

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 [20], this is not an inherent limitation. The virtine runtime could apply a combination of link-time wrapping and binary rewriting to migrate library code automatically to run in virtine context. Others have applied such techniques for software fault isolation (SFI) [55], even in virtualized settings [23].

Serverless and UDFs. 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 constructs a container to achieve the desired level of isolation [54]. 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. A single wasp client (even a container) could be used to handle all requests and spawn virtines as needed, and any external access that is isolated through the use of containers could be emulated by the hypervisor. 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, 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 would be used in real settings, we do not currently integrate with existing serverless platforms or database engines. This integration is currently underway, using OpenFaaS 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 benefit of compiling into a small (~578KB) image, and is easily portable. We envision that with sufficient runtime support—in particular, a port of the C++ standard library—a V8 virtine implementation is certainly feasible. We believe our evaluation demonstrates the potential for incorporating virtines with 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 [14]. This affects start-up latencies when launching virtines, as described in §6.4.

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 CoW mechanisms to reset a virtine, as in SEUSS [16].

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 [12, 30] 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 [33]) 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 serverless. Virtines can be used to change how serverless platforms isolate functions. Traditionally, a serverless function is isolated by spawning a new container for the function, then running the function in a new process for each invocation. This relatively expensive operation could be avoided by creating a single process for each function, and by spawning a virtine for each invocation. By using virtines and a co-designed virtine client, a stateful function within a serverless application might be invoked, and access to resources like S3 can be provided through rich, high-level hypercalls. Virtines can be used to allow for more tunable isolation boundaries within serverless functions.

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. Virtines might also be used to apply security-in-depth to JIT compilers. For example, bugs that lead to vulnerabilities in built-in functions or the JIT's slow-path handlers [48] can be mitigated by running them in virtine context. Polyglot environments like GraalVM [57] 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 [14], 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 [28]. 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 are acceptable. Virtines enjoy several unique properties: they have an easy-to-use programming model, they implement an abstract machine model, allowing 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 [20], which allow for programmer-guided isolation by splitting libraries into their own code, data, and configuration sections within the same binary. Unlike virtines, *Enclosures* operate at the library boundary, using a single VM per application. Hodor also provides library isolation, particularly for high-performance data-plane libraries [24].

While TrustVisor [41] 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 [37]. Virtines give programmers more control over isolated components. Glamdring also automatically partitions applications based on annotations [32], but uses SGX Enclaves which have more limited execution environments than virtines. Erim enforces memory protection using Intel memory protection keys [53].

With Wedge [15], 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 [13]. Unlike virtines, Dune's virtualization is at *process* granularity. Similarly, SMV isolates multi-threaded applications [25]. Lightweight-Contexts (LwCs) are isolated execution contexts *within* a process [34]. 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. Nooks [52], LXD [43], and Nested Kernel [18] all implement isolation for kernel modules.

Virtualization. Wasp is similar in architecture to other minimal hypervisors (implementing μVMs). Unlike Amazon's Firecracker [10] or Google's Cloud Hypervisor [22], 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 [56] (especially the networking interface) and uhyve [31]. 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 [51] 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 [17]. 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 [16, 19, 45]. These systems achieve start-up latencies in the sub-millisecond range with aggressive caching of runtime state, which we also employ.

Execution environments. Jitsu [38] 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 [39], 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 [26, 47].

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 to create and run virtines microsecond start-up latency with limited slow-down. 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 these 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.

Availability

All code for Wasp, our virtine runtime environments, our compiler modifications, in addition to our experimental framework are freely available and will be released upon acceptance.

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] 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

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

[12] 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 (SOSP '03)*. 164–177.

[13] 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.

[14] 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

[15] 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)*. 309–322.

[16] 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)*. Article 32.

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

[18] 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

[19] 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)*. 467–481. https://doi.org/10.1145/3373376.3378512

[20] 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

[21] 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

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

[23] 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 (ICAC 2014)*. 85–96.

[24] 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

[25] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. 2016. Enforcing Least Privilege Memory Views for Multi-threaded 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

[26] Galen C. Hunt and James R. Larus. 2007. Singularity: Rethinking the Software Stack. *SIGOPS Operating Systems Review* 41, 2 (April 2007), 37–49.

[27] Intel Corporation 2016. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B & 3C): System Programming Guide*. Intel Corporation.

[28] 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 (NSDI '19)*. 1–16.

[29] 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 (HotOS XVI)*. 169–173.

[30] 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)*.

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

[32] 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

[33] 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)*. 973–990. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[34] 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 (OSDI '16)*. 49–64. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/litton

[35] 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 (ROSS '19)*.

[36] 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*.

[37] 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

[38] 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

[39] 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.

[40] 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 (SOSP '17)*. 218–233.

[41] 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

[42] 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

[43] 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

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

[45] 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)*. 57—-69.

[46] 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.

[47] 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.

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

[49] 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

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

[51] 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

[52] 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

[53] 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

[54] 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/

[55] 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

[56] 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 (HotCloud '16)*. https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/williams

[57] 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