Hypervisor Dissociative Execution: Programming Guests for Monitoring, Management, and Security

Andrew Fasano*, Zak Estrada*, Tim Leek*, William Robertson†
*MIT Lincoln Laboratory, Lexington, USA
Email: {fasano, zje}@mit.edu, tleek@ll.mit.edu
†Northeastern University, Boston, USA
Email: w.robertson@northeastern.edu

***Abstract*—Both cloud providers and users wish to manage, monitor, and secure virtualized guest systems. This is typically accomplished with custom agent programs that run inside a guest or complex virtual machine introspection (VMI) systems that operate outside a guest. Agents are limited by the need to install and maintain them in each guest, while VMI systems are limited by the need to understand guest kernel internals. We introduce *Hypervisor Dissociative Execution*, or HyDE, a new approach that operates between these extremes to avoid their limitations and provide a robust and flexible mechanism to examine and modify a guest from the outside. In the HyDE model, developers assemble programs that mix out-of-guest logic with in-guest system calls. These programs are launched from outside a guest where they are able to co-opt the execution of guest processes. We present an open-source prototype HyDE implementation paired with 10 HyDE programs that address a wide range of user needs from password resets and guest process enumeration to dynamically generating a software bill of materials. We evaluate the utility, robustness, and performance of HyDE by executing the example programs while concurrently running standard benchmarks within multiple guest systems. Our results show that HyDE maintains system stability and incurs negligible overhead for one-off analyses or modifications. In persistent operation, HyDE incurs overhead as low as 7% in a multi-node cloud application benchmark.**

1. Introduction

Aho and Ullman argue that "Computer Science is a science of abstraction — creating the right model for a problem and devising the appropriate mechanizable techniques to solve it" [1]. In this work, we present Hypervisor Dissociative Execution (HyDE), a novel hypervisor-based programming model that provides powerful abstractions for monitoring and managing the operation of a cloud-based Virtual Machine.

Infrastructure as a Service ("IaaS") establishes a clear division of responsibilities: a provider manages the platform upon which users run an OS and software of their choosing inside a virtualized guest [2]. It also, ideally, provides monitoring and management capabilites which are of value to both users and providers. A number of implementation options exist to achieve these ends. One is to develop an agent and mandate its installation in guests. This solution is not ideal as a compromised tenant could easily circumvent the agent, possibly even from a guest user program. Another option is virtual machine introspection (VMI), which can provide equivalent abilities from outside guests by analyzing their OS memory. However, VMI requires significant expense to prototype and maintain as it must translate the raw state of guest memory and registers into meaningful OS concepts using detailed knowledge of guest internals [3].

In this work, we introduce HyDE, which leverages injected guest system calls as simple building blocks from which to fashion guest monitoring and management without the need for agents or VMI. This use of system calls to bridge the semantic gap is a key innovation, here, serving as a simple high-level abstract interface from which to quickly build monitoring and management gadgets. With HyDE, we simply request the desired information or effect, directly, from the guest, via a combination of well-understood system calls and out-of-guest logic. The gadgets are simple, robust, and typically trivially re-useable across OS versions, despite requiring no detailed OS knowledge beyond the system call interface.

HyDE augments a hypervisor with an event-oriented programming interface that allows injecting syscalls into a guest while analyzing results on the host. Using this interface, developers can create a variety of "HyDE programs." These HyDE programs may query the guest to learn about its state on demand or be run periodically for sampling-based cloud-debugging schemes. Alternatively, they may modify a guest for management purpose (e.g., resetting a password) or to enforce security policies (e.g., blocking root processes from listening on external network sockets). HyDE's programming interface provides abstractions to mask the significant complexity of reliably co-opting the execution of multiple guest processes that arises when working with multicore guests and preemptive multitasking. Fundamentally, HyDE is not tied to a specific operating system nor architecture and is sensitive only to changes to the syscall application binary interface (ABI) and syscall semantics.

We developed a prototype implementation of HyDE with support for the x86_64 Linux syscall ABI and designed

10 sample HyDE programs to enable monitoring, managing, and securing of guest systems. With this prototype, we evaluate the performance and reliability of multiple guest systems running various benchmark and test suites while the guest is monitored and managed by HyDE paired with each of our HyDE programs.

Across tested guests spanning 3 major Linux kernel versions and 2 distributions, we find that HyDE programs successfully accomplish their intended tasks and do not introduce any unexpected changes into the guests. Our results show HyDE programs that make a one-time query or modification add negligible performance impact. For HyDE programs that persistently monitor or manage a guest, we find performance overheads as low as 7.4% in a multi-node cloud performance benchmark.

This paper makes the following contributions:

1) HyDE: A new set of abstractions and a technique for programming a guest from the hypervisor to analyze and modify guest state without the need for understanding or continually tracking low-level guest kernel details.
2) A prototype implementation of HyDE targeted at x86_64 Linux guests.
3) Ten powerful HyDE programs that demonstrate how HyDE can enable monitoring, management, and security of guest systems from a hypervisor.
4) An evaluation of how HyDE and HyDE programs affect behavior and performance across varied guest systems.

2. Background and Motivation

"Cloud computing" as a term encompasses a broad spectrum of models for providing elastic computing resources to users; however, "Infrastructure as a Service" (IaaS) is one of the most common. In IaaS, users manage a guest OS and the software that runs within it. Cloud providers manage the physical infrastructure and use a hypervisor, or a virtual machine monitor (VMM), to concurrently run one or more guests that are isolated from each other [4], [5]. Providers typically utilize hardware-assisted virtualization features which require the guest to be of the same architecture as the host. As such, most virtualized guests are 64-bit x86. KVM [6], Xen [7], and VMware [8] are examples of commonly-used VMMs [9].

In addition to traditional "in-band" interactions where users launch commands from within their running guests, cloud providers typically support management-oriented, out-of-band (OOB) interactions that are initiated from outside of a guest. VMMs typically support some OOB interactions around emulated peripherals; however, the utility of such interfaces depends on an individual guest's configuration. For example, a virtual serial console may not be of value if the guest does not provide a shell on it.

If a user installs a "management agent" within their guest, additional OOB interactions such as executing a command within the guest can be performed [10]. However, agents suffer from three drawbacks: they must be compatible with the user's environment, they must be installed with guest cooperation, and they may be removed or altered by malicious guest programs. Alternatively, with knowledge of guest kernel data structure layout and locations, Virtual Machine Introspection (VMI) [3] enables OOB examination or modification of guest state from a VMM.

As one example, IaaS providers commonly offer users an OOB mechanism to reset their guest's administration account credentials, such as a password or SSH key. According to their public documentation (described in Section D.1), Amazon AWS, Digital Ocean Droplets, and Akamai Connected Cloud all accomplish this by powering off the guest and modifying the boot disk image. This workflow leads to downtime and, as noted by Digital Ocean's documentation, may not work depending on the guest's configuration.

Though clearly valuable, existing OOB tooling only scratches the surface of the functionality that could be provided from a well-designed OOB interface for guest analysis and control. Platforms that provide users the (opt-in) ability to deploy sophisticated custom or pre-built OOB capabilities could gain competitive advantage. Platforms could also make use of OOB interventions in extraordinary situations, e.g., for timely remediation of a widespread, critical, and actively-exploited vulnerability. We believe the HyDE model outlined in this work is an example of such an interface that could be used to enable a wide variety of beneficial OOB capabilities.

3. HyDE: Hypervisor Dissociative Execution

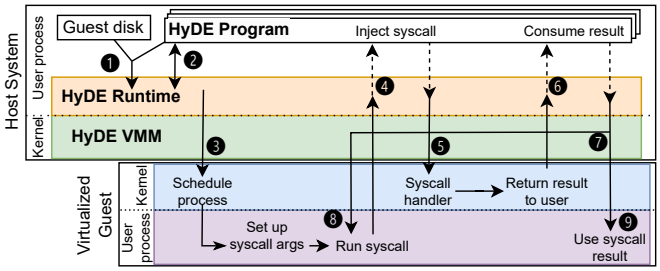

Figure 1. The HyDE runtime and VMM work together to enable user-provided HyDE programs to co-opt the execution of guest processes at the system call interface between the guest processes and guest kernel.

3.1. Assumptions and Threat Model

HyDE runs on a host machine from where it analyzes and injects syscalls into a guest. As such, the host machine is trusted and the guest kernel is partly trusted. Guest applications and users are untrusted. HyDE assumes the details of the guest kernel's syscall interface are known. A malicious guest kernel could violate this assumption and cause HyDE to produce incorrect results, but a malicious kernel could not compromise the HyDE VMM. If a guest kernel is initially trusted, HyDE could be used to restrict privileged operations from user programs that could be used to circumvent its syscall interposition (see *UntrustedRoot* in Section 5).

3.2. HyDE Design

When any guest process issues a syscall, HyDE can co-opt the process's execution by either forcing the guest to run one or more new system calls, modifying the originally requested syscall, or both. This design enables a wide range of solutions to problems that are difficult to solve with existing tools. Our primary insight is that system calls are a relatively stable interface as well as the key events that affect a system's state. HyDE consists of four components:

- **HyDE programs**: Developer-created programs to address specific problems using HyDE
- The **HyDE VMM**: A standard VMM extended to inform the HyDE runtime of guest syscalls and returns
- The **HyDE runtime**: An application to load, unload, and manage execution of HyDE programs
- The **HyDE SDK**: A library for writing HyDE programs

An overview of HyDE is shown in Fig. 1. First, (1) a user asks the runtime to begin execution of a guest system. The user may specify one or more HyDE programs to load, or programs may be provided after a guest has started. (2) The runtime loads the specified HyDE program and is told which syscalls that program wishes to co-opt and how. (3) The runtime instructs the VMM to use CPU virtualization to run the guest, but trap to the VMM when syscalls are issued and returned from. The guest then begins or continues its execution. Eventually, (4) a guest process issues a syscall that traps to the VMM which informs the runtime of the syscall. If a HyDE program has requested to co-opt the pending syscall, the runtime allows that program to potentially inject a new syscall instead. The syscall is injected by the runtime and VMM which modify the guest state to replace the original syscall's details with those of the injected syscall. After modification, (5) the guest OS services the syscall. When it finishes, it again (6) traps to the VMM which invokes the runtime. The runtime informs the HyDE program of the syscall result and the runtime may (7) decide to inject another syscall or finish its execution. (8) If a new syscall is to be injected, the runtime and VMM modify the guest such that the original syscall instruction will be run again and the process repeats from (4). Alternatively, if the HyDE program is finished injecting logic, (9) the guest process is allowed to resume execution from immediately after the original syscall instruction.

3.3. HyDE Programs

Developers create HyDE programs by writing "co-opter" functions. These co-opters are asynchronous functions that execute on the host machine and are instantiated and managed by the HyDE runtime before guest syscalls are executed. A key innovation in HyDE's design is how co-opters provide an abstraction for developers to co-opt processes at various scales while taking actions in multiple execution contexts.

When a guest process issues a syscall that is to be co-opted, the HyDE runtime instantiates a new instance of the specified co-opter and begins its execution before the guest is allowed to continue. The co-opter is provided with details of the syscall that triggered it and may report or consume information from its primary execution context, the host system. The co-opter may also elect to run a syscall inside the execution context of the guest process that triggered it. To do so, it yields a *syscall object* to the HyDE runtime. The runtime then modifies the guest state such that the specified syscall will be run, and resumes execution of the guest process.[1] When the injected syscalls returns, the HyDE runtime extracts the return value and resumes the appropriate co-opter instance. With this design, HyDE programs can co-opt processes at various scales and take actions in multiple execution contexts. Unlike other approaches, HyDE programs do not require low-level knowledge of guest kernel details, only an understanding of its syscall interface. For many OSes, this interface is well-documented and stable.

```
async PreListen(ctx):
  uid := yield syscall(getuid)
  if uid == 0:
    SocketInfo S
    yield syscall(getsockname, ctx.args[0], S)
    if is_remote_socket(S):
      return -EPERM // Skip original syscall
  yield ctx.orig_syscall // Run original syscall
```

Figure 2. Pseudocode for a HyDE co-opter that blocks root processes from listening on remote sockets. The co-opter executes in both the host and guest context, and independent instances may concurrently co-opt multiple guest processes.

Case Study: Low-Privilege Services. Consider the following security policy that a user may wish to enforce on their system: *No process owned by the root user may ever listen on a remotely-accessible network socket*. Such a policy could be enforced by a customized kernel, a kernel module configured and built to work with the user's system, a syscall sandbox such as seccomp, or a number of other options. However, each of those options requires invasive modifications to the guest. Alternatively, a user could enforce this policy by running a HyDE program from outside the guest with a co-opter registered to run whenever a guest process issues the `listen` syscall. A pseudocode sketch of such a co-opter is shown in Fig. 2 which uses syscall injection to identify the current user and examine the state of the socket that is to be listened on. Depending on its analysis, the co-opter may allow the syscall to be run or return an error code to the guest process.

1. A guest process may issue multiple types of the syscall that triggered the creation of a co-opter. Although an instance of that co-opter already exists, the HyDE runtime identifies this as an independent execution context in the guest and instantiates a new, independent instance of the co-opter to handle the new event. This next instance of the co-opter has independent state from the first and may run a different code path.

3.4. HyDE Virtual Machine Monitor

HyDE requires its VMM to provide two primitives to the runtime: a synchronized syscall event stream and an interface for reading/writing guest state. The stream consists of two classes of events: *syscall entry* and *syscall exit*. At each event, the runtime may read or write guest registers or memory if it so chooses and instruct the VMM to resume execution of the virtual CPU that triggered the event.

3.5. HyDE Runtime

Users interact with the HyDE runtime to start and stop virtualized guests and to load or unload HyDE programs. The runtime is responsible for scheduling HyDE co-opters in response to the syscall event stream from the HyDE VMM. It is also tasked with mapping syscall exit events to their corresponding syscall entry event, a challenge that required solving substantial engineering problems that heretofore have not been discussed in the literature, which we describe in Section 4.1.

On syscall entry. The HyDE runtime examines the register state to identify if this syscall was triggered by a HyDE co-opter or a guest process. If the event was triggered by a guest process and the specific syscall is registered to be co-opted, a new co-opter is instantiated. That co-opter executes until it provides a syscall to inject. If the event was triggered by an existing co-opter, the runtime loads the parameters of the syscall to be injected. In either case, the runtime modifies the guest state such that the provided syscall will be run and resumes execution of the guest process. If neither condition is met, the guest system call is run as normal.

On syscall exit. When the kernel finishes processing a syscall, the HyDE runtime examines the guest state to determine which, if any, HyDE co-opter injected the just-returned syscall. If such a co-opter exists, the runtime provides it with the return value of the injected syscall and resumes execution of the co-opter until it either yields a syscall or terminates. If the co-opter terminates or the syscall was not injected by a co-opter, the guest process resumes as normal. Otherwise, the co-opter specifies another syscall to inject. In this case, the HyDE runtime modifies guest state such that the next instruction in the co-opted process will again be the syscall instruction.[2] Upon termination, a co-opter may request for the runtime to unload the HyDE program that spawned it.

3.6. HyDE Programming Interface and SDK

HyDE programs are written in C++, compile into shared libraries, and are loaded by the HyDE runtime. HyDE programs are required to export an *init_plugin* function which associates syscalls with co-opter functions. The co-opter functions are C++20 asynchronous coroutines that yield syscall objects to execute within the guest.

```
#include <sys/sysinfo.h>
#include "hyde_sdk.h"

SyscallCoroutine guest_info(SyscallCtx* ctx) {
  struct sysinfo i;
  yield_syscall(ctx, sysinfo, &i);
  printf("Uptime %lu\n", i.uptime);
  printf("# processes: %d\n", i.procs);
  printf("Load (1/5/15 min): %lu %lu %lu\n",
    i.loads[0], i.loads[1], i.loads[2]);
  yield_and_finish(ctx, ctx->pending_sc(),
  ExitStatus::FINISHED);
}

bool init_plugin(CooperMap map) {
  map[-1] = guest_info; // Before any syscall
  return true;
}
```

Figure 3. *GetSysInfo* HyDE program to inject sysinfo before the next guest syscall and report the results to the host. After the injected syscall returns, the program unloads itself by returning an ExitStatus of *FINISHED*, and the guest program runs the original syscall.

Fig. 3 shows a small HyDE that extracts information about guest state. This HyDE program injects the `sysinfo` syscall before the next syscall is issued by any guest process. The *host* buffer `i` is passed as an argument to `yield_syscall` which injects the syscall in the guest and manages memory synchronization. After the injected syscall returns, the program prints the result on the host, configures itself to unload, and runs the originally requested syscall.

While powerful HyDE programs can be created directly atop the HyDE interface, the HyDE SDK includes three key features to greatly simplify development of HyDE programs.

Yield from another coroutine. The SDK provides a macro, *yield_from*, that allows HyDE programs to yield execution to another coroutine that can inject syscalls into the guest.

Memory access helpers. The SDK provides coroutines for reading and writing both fixed-size buffers and null-terminated strings in guest memory. These helpers are built upon the reliable guest memory access techniques described in Section 4.2.

Transparent argument synchronization. Many syscalls take arguments and/or return values through pointers to structures or arrays. A co-opter may explicitly manage these input and output results by injecting a syscall to allocate a buffer of guest memory, manually populate the buffer, and then inject a syscall with arguments pointing to various addresses in that buffer. After the syscall runs, the co-opter could then copy the results out of guest memory and deallocate the buffer with another injected syscall. This process is significantly more complex than standard C program development, where stack- and heap-based arguments can easily be passed to functions and directly accessed.

The HyDE SDK provides *yield_syscall*, a macro for injecting a syscall into a guest while transparently syn-

2. To properly support signal handling logic which may trigger before the subsequent instruction in the co-opted process, these modifications employ a strategy similar to our state storage technique presented in Section 4.1.

chronizing input and output arguments between host and guest memory. The macro automatically injects the necessary syscall to allocate and copy input arguments into guest memory. The user-requested syscall is then injected into the guest, with host pointers replaced by pointers to the appropriate guest address that was just populated. After the injected syscall finishes, the value of each argument is copied back out of the guest's memory and the corresponding host variables are updated, so long as they are not marked as *const*. Subsequent injected syscalls can reuse this allocated guest memory which is freed when the co-opter terminates.

3.7. Safety of Co-Opting Guest Processes

For HyDE to be of value, users must be confident that running a HyDE program will not unexpectedly cause guest processes to deviate from their normal behavior. At the same time, users may deploy a HyDE program that makes an arbitrary modification to their guest as specified by the program's developer. To consider the implications of how HyDE can affect a guest's behavior, we first distinguish between two types of co-opters that HyDE programs may include: *state-preserving* and *state-altering*. A state-altering co-opter injects syscalls or modifies registers/memory, explicitly modifying guest state. On the other hand, a state-preserving co-opter only injects syscalls that are not designed to alter guest state.

Although executing a state-preserving co-opter does not explicitly modify guest state, the execution of a co-opter may be detected through kernel customization, cross-process introspection (e.g., *ptrace*), or side channels. As HyDE is detectable, we cannot guarantee that guest behavior could never be altered by running a state-preserving co-opter. However, when running "reasonable" guest systems that do not alter their behavior based on cross-process introspection or side channels, state-preserving co-opters will not cause a guest system to deviate from its normal execution.[3]

Unlike state-preserving co-opters, state-altering co-opters may cause reasonable systems to deviate from their normal behavior. While this means that these co-opters are capable of reducing a guest's stability, it also means that they can enable a wide range of useful applications such as resetting a password, modifying a file, or changing a network configuration. Developers of state-altering co-opters should precisely understand how the altered state may impact a guest and communicate those impacts to users.

4. HyDE Implementation

We created an open-source HyDE prototype using QEMU/KVM [11], [6] with support for virtualized x86_64 guests and a C++ interface for HyDE programs. The implementation instantiates three of the design-level components described in Section 3: a HyDE VMM for detecting syscall events in a virtualized guest, a HyDE runtime for managing and controlling HyDE programs, and a HyDE SDK to assist in writing HyDE programs.

The HyDE VMM extends the Kernel Virtual Machine (KVM) kernel module from Linux 6.2.2 to detect `syscall` and `sysret` instructions. These modifications were a straightforward extension of Pfoh et al.'s technique for Intel VT-x. Specifically, the syscall extensions bit is cleared to force 64-bit x86 guests to trap to the VMM on each `syscall` and `sysret` instruction where those instructions can be logged and emulated [12]. Beyond the logic to trap on the `syscall` and `sysret` instructions, we added a mechanism to toggle this mode and to synchronously inform the HyDE runtime when those instructions are to be executed. The KVM modifications total 299 lines of code. The HyDE runtime is built atop QEMU 7.2 which interacts with the VMM to launch guest systems and manage HyDE programs. This runtime, implemented in 2,704 lines of code, allows users to load and unload HyDE programs while a guest is running. Internally, the runtime manages initialization and execution of the co-opters within each HyDE program. Finally, the HyDE SDK is implemented in 1,408 lines of C++ code.

During implementation, we had to develop novel solutions for reliably tracking syscall state and allowing a VMM to access guest virtual memory. We describe these in the remainder of this section.

4.1. Reliably Tracking Syscalls

Detecting `syscall` and `sysret` instructions for virtualized guests is a well-studied problem with numerous solutions [13], [14], [15], [12]. The next challenge is to map each `sysret` back to the `syscall` for which it is returning. A guest system may be concurrently running various processes, potentially with multiple threads, spread across multiple cores. At any point in time, multiple threads may be waiting for syscalls to return. While a syscall is pending, a thread may be preempted by the kernel to run a signal handler or another thread may be scheduled to run. As such, mapping return values back to the correct syscall is non-trivial. The literature describes different approaches for this mapping using various identifiers such as the guest instruction pointer [14], the address space ID (ASID) for x86 guests [12], or a combination of the two [15].

Through experimentation, we found these existing techniques to be unreliable as multiple syscalls may be in-flight concurrently in different threads with identical identifiers or even within a single thread. For example, multiple threads may execute a syscall at the same program counter before another returns. Or a process may be preempted while executing a syscall, and a signal handler may execute a second syscall before the original syscall returns. On multicore guests, processes may be migrated between cores while executing a syscall. The ASID cannot be used to track such syscalls as it may change when a process resumes on its new core.

To properly handle complex programs and multicore systems, we devise a new "state-storage" scheme for mapping

3. We base this claim on the stability evaluation performed in Section 6.

each `sysret` back to its corresponding syscall. When a guest process issues a syscall that is co-opted by a HyDE program, HyDE reads and replaces four guest registers before the `syscall` instruction is executed. In these registers, HyDE writes a magic value indicating a potentially co-opted syscall, a unique identifier for that syscall, the address of the `syscall` instruction, and a hash of the identifier and instruction address. Those values are placed in callee-saved registers that are not inputs to the `syscall`. Thus, the kernel's logic will not be affected by the register changes, and those registers will be restored before returning to user space via `sysret`. At that `sysret` instruction, HyDE examines the register values to identify which syscall is returning. Notably, this approach supports tracking syscalls across processors and remains correct in the presence of signal handlers.

With the ability to reliably map each `sysret` back to a corresponding syscall, HyDE can identify when an injected syscall returns and update the state of both the guest and the HyDE program that requested the injection.

4.2. Reliably Accessing Guest Memory

Although a VMM has access to a guest's physical memory, reliably reading and writing to guest virtual addresses (GVAs) is challenging as the mapping between GVAs and guest physical addresses (GPAs) is managed by the guest. VMMs may walk page tables to recover GVA to GPA mappings, but these passive approaches may be insufficient for analyses that wish to access paged out guest memory or to write to a GPA that aliases to multiple GVAs.

As part of the HyDE interface, we designed two primitives to enable analyses to reliably read and write GVAs from a VMM. Our first primitive forces the guest to page in a target address by injecting a syscall that makes the kernel to examine the target address. This is accomplished with the `access` syscall which attempts to read a pointer, then does some additional checks which are irrelevant to this use case. A return value of `-EFAULT` indicates the address is invalid. Any other return value indicates that the kernel successfully read, and thus paged in, the memory at the given address.

The second primitive for reliable guest memory access arises from a complication caused by demand paging in the guest OS. This complication involves data that is initially shared across processes (e.g., anonymous memory allocation, copy-on-write during `fork`). For example, when a process allocates memory via a call to `mmap` with the `MAP_ANONYMOUS` flag, it receives zero-filled memory. Modern kernels perform this allocation lazily: although multiple distinct GVAs may be given to processes, those distinct GVAs will be aliases for the same GPA which is filled with zeroes and is marked as non-writable. This allows the guest kernel to allocate physical pages when needed at the first write. However, when a VMM translates a GVA and directly accesses the corresponding GPA, it may be accessing a shared page used by multiple distinct GVAs. Reading from this page will return the correct data, but writing to it will write to the shared page, effectively modifying the memory of all the GVAs that alias the GPA.

To solve this problem, HyDE again uses syscall injection. Before HyDE allows a HyDE program to write N bytes to a GVA, it first injects a call to `getrandom` to write N bytes of pseudorandom data (from `/dev/urandom`) to the destination GVA. This forces the guest to allocate a new physical page for the GVA if necessary, making it safe to write to the GVA as it will be backed by a unique GPA.

5. HyDE Programs

In this section, we explore the range of problems HyDE programs can address, introduce ten HyDE programs we have created, and describe the design of two in detail. Table 1 summarizes these programs, listing their name, description, source lines of code (SLOC) [16], lifetime, and whether they modify guest state.

HyDE programs can examine and modify guest state from the VMM to address a variety of problems. *One-off* programs complete one task and then terminate, while *persistent* programs continue until deactivation by a user. Unlike one-off programs, persistent programs may introduce non-negligible performance overheads to the system.

As introduced in Section 3.7, HyDE programs can also be classified as either *state-preserving* or *state-altering* with respect to guest state. Within the category of state-altering programs, we identify three distinct classes. The first class are programs that leverage knowledge about interfaces used by guest processes to make modifications that are within the bounds of what the interface could reasonably provide. For example, the *EnvAdder* program adds a user-specified value into the environment of newly created processes. The second class are programs that make an explicit modification to a guest based on a user's request. For example, the *PWReset* program modifies the `/etc/shadow` file to change a guest's root password. So long as the behavior of these types of programs is communicated to users, the user bears responsibility for making reasonable modifications to their system. The final category of state-altering programs are those that contain developer errors. Such programs may make subtle modifications to guest state causing guest processes to receive unexpected results from the interfaces they interact with. For example, if a new file was opened and not closed, subsequently opened file descriptors would be incremented by one. While state changes such as these may not always be fatal, reasonable guest processes could deviate from their expected behavior in unexpected ways when such changes are present. Unfortunately, distinguishing between buggy and bug-free programs is a difficult task. We leave this responsibility to the developers who should use standard software engineering practices to ensure that their HyDE programs behave correctly.

Case Study: DSBOM. A problem cloud users face is understanding what software runs on their system. A "software bill of materials" (SBOM) specifies the executables and libraries that are run by a system. Equipped with an SBOM,

TABLE 1. EXAMPLE HYDE PROGRAMS FOR MONITORING, MANAGING, AND SECURING GUEST SYSTEMS.

	Name and Summary	**SLoC**	**Lifetime**	**State Altering**
Monitor	**DSBOM**: Calculate and report hashes of loaded code	167	Persistent	No
	FileAccessLog: Log all files accessed by a guest	41	Persistent	No
	GetSysInfo: Query system uptime and other info from guest kernel	17	One-off	No
	HyperPS: List current running processes	95	One-off	No
Man.	**PwReset**: Change password hash in *shadow* file	81	One-off	Yes
Secure	**2faRoot**: Require users to enter code before escalating privileges	144	Persistent	Conditionally
	EnvAdder: Conditionally add environment variable to processes	80	Persistent	Conditionally
	NoRootSock: Prevent root users from listening on remote sockets	30	Persistent	Conditionally
	SecretFile: Conditionally allow guest to read a host file	127	Persistent	Conditionally
	UntrustedRoot: Prevent privileged actions for all users	124	Persistent	Conditionally

a user can identify known-vulnerable components of their system and plan remediation or patching strategies [17].

DSBOM (Dynamic SBOM) is a HyDE program that logs the paths and hashes of each executable and library loaded and run by a guest system. This program co-opts guest processes as they issue the `execve`, `execveat`, or `mmap` syscalls. At the `execve` family of syscalls, DSBOM opens the target file and reads it in chunks to compute its hash. The file name and hash are logged outside the guest. At the `mmap` syscall, the program checks if the guest is trying to map data into memory from a file descriptor (e.g., a shared library). If so, it uses the `readlink` syscall on the `/proc/self/fds/<FD>` file to determine the path to the file, then hashes and logs it as before. Variations on this HyDE program could be created to proactively scan a file system to compute this material or to enforce a security policy where components and their hashes are checked against an existing SBOM before they are allowed to be loaded by a guest.

Case Study: UntrustedRoot. Cloud guests are often Internet-facing and long-lived, thus making them attractive targets for malware that will install and hide itself using a rootkit [18]. The *UntrustedRoot* HyDE program is designed to be loaded after a system has booted and reached a steady state. Once launched, *UntrustedRoot* enforces a security policy where no guest users, including root, can perform certain actions that are commonly used by rootkits, such as loading new kernel modules, overriding the dynamic linker, accessing most device files, and accessing kernel memory from user space (i.e., `/proc/kcore`) [19]. This is accomplished by co-opting four syscalls to always return `-EPERM` and another three to examine arguments and conditionally block the operation if it is deemed unsafe.

6. Evaluation

Our first set of experiments validates a fundamental requirement of HyDE: that it is possible to inject syscalls into a guest without negative side-effects. After establishing this, we test the correctness and portability of our ten example HyDE programs by running them with multiple guest systems. We additionally check for correctness within the guests by ensuring they pass a complex test suite as expected. Finally, we conduct a performance evaluation of HyDE, measuring how syscall injection frequency and specific HyDE programs affect guest performance across multiple workloads.

6.1. Test Configuration

All experiments were conducted on a 16-core Intel Xeon E5-2637 v3 CPU host machine with 377 GiB of memory running Ubuntu 22.04.2 LTS and Linux 6.2.2. We run virtualized guests with different major kernel versions and vary memory and CPU to run HyDE in multiple configurations: **Ubuntu 23.04.02 LTS** with Linux 6.2.0-20-generic, 8 cores, and 2 GiB of memory; **Ubuntu 22.04.02 LTS** with Linux 5.15.0-67-generic, with 8 cores, 1 GiB of memory; and **CentOS Stream 8** with Linux 4.18.0-489.el8.x86_64, with 4 cores, 4 GiB of memory.

To evaluate HyDE's effect on *guest* behavior and performance, we use a combination of a syscall-intensive test suite, a syscall microbenchmark, an HTTP server benchmark, and a standard cloud workload. We select the GNU Coreutils test suite [20] to measure if HyDE and HyDE programs would inadvertently alter guest behavior. We select version 9.3 for use in both Ubuntu guests and version 8.30 for the CentOS guest as newer versions did not compile on that system. This test suite dynamically selects appropriate tests to run depending on guest configuration. Different tests are selected between the three guest systems, and we observe minor variance even within identical configurations. We refer interested readers to Section C for more details.

We measure the per-syscall overhead with lmbench's syscall microbenchmarks [21] and evaluate a syscall intensive workload using the Coreutils test suite. To estimate HyDE's performance impact on cloud applications, we use the wrk2 HTTP latency benchmark[4] and the Graph-Analytics test from CloudSuite 4.0 [22], [19].

6.2. State-Preserving Syscall Injection

To test our claim that HyDE can inject syscalls into a guest while preserving correct guest execution, we developed a simple HyDE program, *PerfEval*, that injects the `getpid` syscall after every N observed syscalls. This syscall is side-effect free: `getpid` returns the current process ID and should make no other modifications to the process that runs it. After `getpid` completes, *PerfEval* runs the

4. https://github.com/giltene/wrk2

original syscall in the co-opted process. We run the Ubuntu 22.04 guest with *PerfEval* loaded and $N \in \{1, 100, 1000\}$ 10 times each.

All of the run Coreutils tests pass in these three configurations. These tests not only validate HyDE's fundamental requirement of state-preserving syscall injection, they also demonstrate the reliability of HyDE's syscall tracking mechanism presented in Section 4.1. That is, if HyDE were to incorrectly track syscall state, the originally-requested syscalls invoked after `getpid` could be executed in the wrong context and would trigger failures in the test suite.Furthermore, we validate that the `getpid` syscall is being injected into guest processes using `strace` inside the guest.

6.3. Correctness and Portability

To evaluate if HyDE and HyDE programs behave correctly without making detrimental modifications to guest state, we execute the Coreutils test suite in the presence of each of our 10 example HyDE programs described in Section 5 across the three different guest OS/virtual machine configurations. With each of these HyDE programs, we design unit tests that run commands in the guest and examine both input and output of guest results to validate that the program is behaving correctly. Section A describes the testing strategies used for each program. For experiments with persistent HyDE programs, we run the unit test twice, with the Coreutils test suite run between them. For experiments with one-off HyDE programs, we launch the Coreutils test suite and load the HyDE program in the middle of its execution. We then check if the HyDE program behaved correctly when the Coreutils test finishes. For each experiment, we check if all Coreutils tests pass and if the HyDE program unit tests pass.

We collect the Coreutils test suite results with each of our HyDE programs, HyDE with no programs loaded, and stock KVM. Each experiment is repeated 10 times and the results are identical: **All Coreutils tests pass as expected, and all HyDE program unit tests pass for all guests**. When running the *EnvAdder* HyDE program, one expected "failure" reliably occurs: the `misc/env-S` test, which validates the process's environment against expected values. This test failure demonstrates that *EnvAdder* is, in fact, working as intended.

The HyDE programs function correctly across all three guest systems without reducing guest stability. Although it is beyond the scope of this paper, we have previously built a HyDE implementation atop an emulator where we created HyDE programs that were portable across multiple architectures, endiannesses, and even some OSes. We refer interested readers to Section B.

6.4. Performance Overhead

Now that we have established that HyDE and HyDE programs can be used to add functionality to a guest without reducing its stability, we evaluate the performance overhead

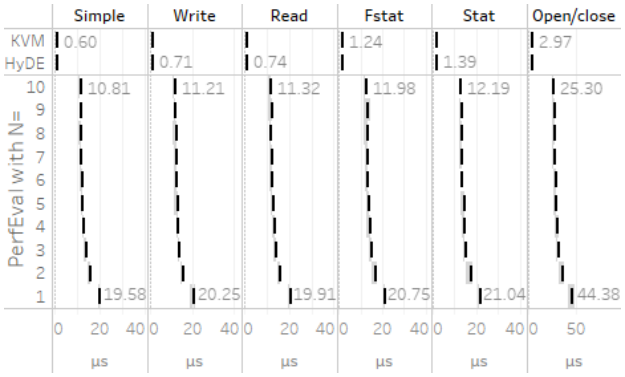

Figure 4. **Guest syscall microbenchmark**: Syscall execution times reported by lmbench and averaged across 10 runs in our Ubuntu 22.04 guest. Stock KVM compared to HyDE's VMM with no HyDE programs and with *PerfEval* injecting a syscall between each of $N \in [1, 10]$ guest syscalls.

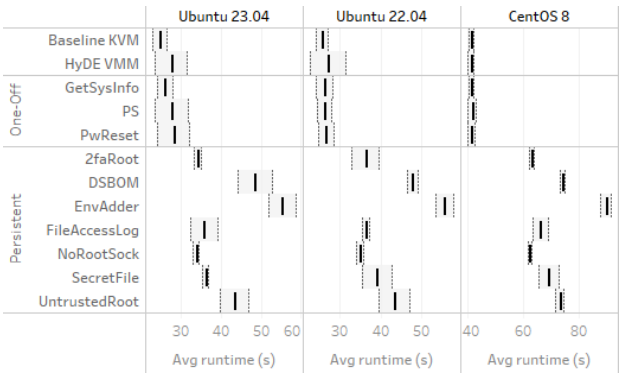

Figure 5. **Coreutils test performance** under stock KVM and HyDE with various HyDE programs across three guests.

of HyDE. We first examine the per-syscall overhead and then measure how this affects overall system performance.

With the Ubuntu 22.04 system, we use lmbench to measure the performance of individual guest syscalls when running HyDE and *PerfEval* with $N \in \{1, 10\}$, HyDE with no HyDE programs, and stock KVM. The results of running this microbenchmark 10 times for each configuration are shown in Fig. 4. When HyDE has no HyDE programs active, its performance is virtually identical to that of stock KVM. However, when running with HyDE programs, this microbenchmark shows a per-syscall overhead ranging from 8.5 to 33x that of stock KVM.

To measure the impact of HyDE on a syscall-intensive workload, we measure the runtime of the Coreutils test suite for all three guest systems. We measure Coreutils test runtime with each of our HyDE programs enabled, HyDE with no HyDE programs, and stock KVM. The results of running this benchmark 10 times for each configuration are shown in Fig. 5. Running the HyDE VMM by itself or with one-off HyDE programs introduces minimal overhead while persistent programs introduce overhead ranging from $36\% - 216\%$, depending on the program.

To estimate the impact of HyDE's syscall analysis and injection on networked applications, we utilize a middle

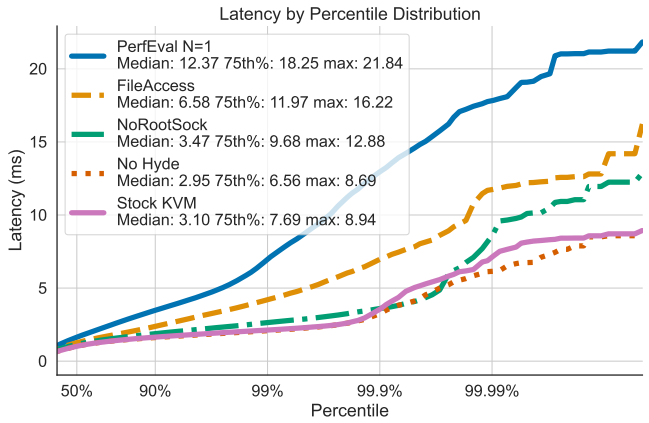

Figure 6. A plot of the high dynamic range histogram generated by the wrk2 benchmark, emphasizing tail latencies. The legend indicates the median and the 75th percentile, as well as the maximum latency observed. The benchmark was run for 4000 requests/s for 60 s.

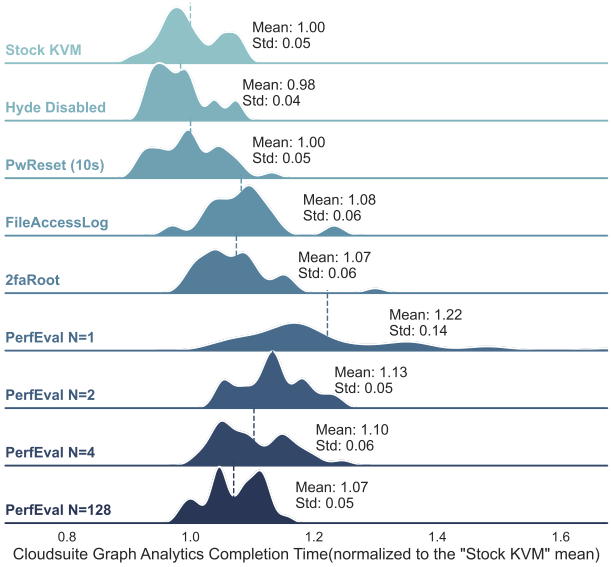

Figure 7. CloudSuite 4.0 Graph Analytics Benchmark Results, 30 samples in each distribution. Distributions normalized to a cluster of virtual machines running with stock KVM (mean execution time: 80 s). "Perf Eval" labels correspond to executions run with *PerfEval* injecting a syscall between every N guest syscalls. "Hyde Disabled" is the HyDE VMM with no programs loaded. Other labels correspond to their respective HyDE program, with "10 s" indicating the one-off "PwReset" program was executed every 10 seconds.

ground between a useful workload and a microbenchmark: the wrk2 HTTP latency benchmark. We run wrk2 against the default nginx server, serving its default `index.html`. Our test system is a single guest running Ubuntu 22.04 with 4 cores and 8 GiB of memory, and we run wkr2 from the host OS. The wrk2 benchmark is designed to stress-test web servers by generating a significant load from a small number of threads. We run wrk2 with its default setting of 100 simultaneous connections but double the number of threads from 2 to 4, constant throughput from 2000 requests/s to 4000 requests/s, and run duration from 30 s to 60 s to increase load. We run the HTTP guest with stock KVM, *PerfEval* with $N = 1$, and two HyDE programs a user may consider running on a web server (*NoRootSock* and *FileAccess*). In Fig. 6, we see that the more expensive *FileAccess* HyDE program has a stronger impact on latency than *NoRootSock*. This is expected as *FileAccess* reads guest memory and writes to the host disk during its execution. We observe that the tail latencies are higher for HyDE programs, but a persistent program with a network security application like *NoRootSock* demonstrates acceptable performance compared to baseline. *PerfEval* has the worst performance as it injects a superfluous syscall alongside each application syscall.

To evaluate the impact of HyDE on a realistic cloud workload, we run the Graph-Analytics test from the CloudSuite benchmark. We run the dockerized version of the benchmark with the default docker seccomp profile, demonstrating no conflict between HyDE and seccomp or HyDE and Docker. In this configuration, we run four co-located instances of the Ubuntu 22.04 guest connected via a virtual bridge. To meet the benchmark's recommended hardware requirements, we configure each guest with 1 core and 20 GiB of memory. To measure worst-case performance impact on this workload, we run the PerfEval HyDE program with $N = \{1, 2, 4, 128\}$ as well as the more realistic, but still performance intensive, *FileAccessLog* and *2faRoot* HyDE programs. As an exaggerated test of a user interacting with a one-off HyDE program during an intense workload, we run the "PwReset (10 s)" test which resets the user's password every 10 s. In all tests with HyDE programs, we enable those programs for all four guests in the cluster and run a baseline on the HyDE VMM with HyDE disabled as well as with stock KVM. Fig. 7 shows that the mean overhead is $\sim 22\%$ for injecting a syscall before every guest syscall. For the more realistic *FileAccessLog* and *2faRoot* programs, the overhead is $\sim 9-10\%$. For *PerfEval* we compare $N = 128$ with $N = 4$ which shows that the performance impact of syscall injection (as opposed to tracking) quickly tapers off when syscall injection is less frequent. We observe a negligible impact to mean execution time with the exaggerated one-off experiment of executing *PwReset* every 10 s test.

Although HyDE introduces high overhead to the runtime of syscalls, the majority of a typical guest's execution is not spent running syscalls. The CloudSuite benchmark results show the overhead of standard HyDE programs may be as low as $\sim$ 7% on a realistic cloud workload.

6.5. Aside: Reliable Virtual Memory Access

To demonstrate both the prevalence of paged out guest virtual memory and the reliability of HyDE's solution described in Section 4.2, we run a variation of our *FileAccessLog* program that attempts to read file paths first using the standard memory translation interface provided by KVM and then using our reliable virtual memory access solution.

The program measures the total number of calls to the `open` and `openat` syscalls, the number of times the standard memory translation interface fails to translate a path name pointer, and if our solution ever fails.

Across 10 runs of the experiment, we see a total of 3,547,787 calls to these two syscalls and 54,403 path name pointers that could not be translated by the standard memory translation interface. While a success rate of 98.47% is high, failing to read some opened file names could be a fatal flaw for a monitoring system. Fortunately, when using HyDE's reliable memory access solution, we never fail to translate a path name pointer and successfully read all file names.

7. Discussion

7.1. HyDE versus Guest Agents

IaaS cloud providers offer their own security monitoring solutions based on guest agents as described in Section D.2. In addition to standing on its own, we see the robust programming model and modularity of HyDE as a valuable contribution to those existing systems, potentially replacing some of those guest agents. For example, the *DSBOM* HyDE program provides information similar to vulnerability audits performed by those guest agents. HyDE could be used to provide and maintain trusted logs outside of the guest.

As HyDE builds off the ability to modify and inject syscalls into a guest from the hypervisor, it is fundamentally less powerful than a kernel-based guest agent which has direct access to guest kernel internals and kernel functions. However, HyDE programs can accomplish many of the same tasks as guest agents and have several advantages in how they are designed and deployed.

Unlike standard, kernel-based guest agents, HyDE programs do not directly interact with internal kernel functions, instead they operate solely on the well understood and stable syscall interface. When any guest process attempts to take some syscall-based action, the HyDE program can examine syscall arguments, run additional or alternative syscalls or a modified version of the original syscall.

HyDE programs are compiled for a host machine (which is expected to be more homogeneous than a collection of guests), and can be deployed on demand across guest kernel versions.

7.2. Benefits of the HyDE Paradigm

HyDE programs can provide many of the same results as guest agents but with four key advantages. **Dynamic deployment**: As HyDE programs operate at the VMM level, they need not be installed prior to use as the VMM can enable HyDE programs as needed. **Decoupled from kernel internals**: Unlike traditional kernel modules that are built against a specific kernel version, HyDE programs compile for a host machine and operate without modification across various guest kernel versions. **Logic is opaque to guest**: Unlike in-guest agents, a malicious guest cannot directly access nor examine the logic of HyDE programs. As such, HyDE programs can be used to store sensitive information such as mitigations to non-public vulnerabilities known by an IaaS provider. **Novel security boundaries**: Co-option of guest syscalls creates novel security boundaries and allows for cloud providers and users to adopt new security models. As seen in the *2faRoot* and *SecretFile* examples, one can keep arbitrary information (e.g., cryptographic keys, proprietary program logic) outside the guest. The VMM could also introduce temporal restrictions to prevent certain guest actions (e.g., as part of a change management process).

7.3. Observing and Circumventing HyDE

As with other VMM-based techniques, HyDE may be detected by a guest. HyDE is visible to kernel-based infrastructures such as ftrace [23] and eBPF [24], in addition to timing side channels and other side effects from injecting syscalls. Visibility is not a new threat to VM monitoring and obfuscation was not a design goal for HyDE. While a guest could detect the presence of HyDE, it would be difficult for a guest to reverse engineer the logic of a HyDE program as the program's logic lives outside the guest and only injected syscalls are observable. Furthermore, a (readily observable) HyDE program could be deployed to block guest processes from using interfaces that reveal the details of syscalls being run.

When working with a benign guest kernel, HyDE can detect, understand, and inject new syscalls into a guest. In such an environment, all processes that interact with the guest kernel through the syscall interface are subject to analysis and modification by HyDE. However, privileged processes may try to circumvent HyDE by either avoiding or reconfiguring the syscall interface. For example, an attacker could load a custom kernel module that adds a non-standard interface for running syscall logic. Once such a modification is made to a a guest, HyDE can be bypassed. However, to make such a modification, the syscall interface must first be used (e.g., to load a kernel module or access kernel memory) and such interactions could be analyzed and prevented with a HyDE program such as the *UntrustedRoot* example.

7.4. Safety of HyDE

A malicious guest could alter its syscall interface to interfere with HyDE programs, but such an attack would only invalidate analysis results and not compromise the host system. More significantly, a buggy or malicious HyDE program could cause a guest to misbehave. As such, developers of HyDE programs should subject them to standard software testing methodologies before deployment and cloud providers should restrict the ability to run HyDE programs to trusted users.

7.5. Performance

As discussed in Section 6, HyDE's overhead is negligible for one-off analyses and transient monitoring. Persistently analyzing all system calls is possible, but the overhead

could be high for syscall-intensive workloads. That said, performance-oriented memory and CPU-bound applications are minimally affected as those applications view syscalls as expensive and attempt to minimize syscall use. Alternative implementations of HyDE could sacrifice generality for performance by pushing some logic into the guest.

7.6. Alternative Implementations

While our implementation of HyDE uses a custom VMM to track and inject guest syscalls, this could also be accomplished with guest cooperation (e.g., a custom kernel module, a kernel debugger configured to break at a guest's syscall handler). It could even be collected *within* a system using built-in syscall tracing and state modification capabilities (e.g., ptrace). Such an implementation would require additional development effort to support diverse guest systems.

8. Limitations and Future Work

Our HyDE implementation demonstrates the value and power of a syscall-based interface for examining and modifying state from outside a guest but there are limitations and areas for future work. Like all programs, HyDE programs may contain logic flaws that lead to unwanted behavior. We have made no effort to formally verify the correctness of HyDE programs as we found manual testing sufficient to ensure correctness of our example programs. More complex HyDE programs may require the use of automated static and dynamic analysis tools.

If a guest kernel restricts the ability for a process to issue syscalls (e.g., with seccomp), subsequently co-opting that process with a HyDE program may introduce undesired behavior if prohibited syscalls are injected.

As HyDE is built around the syscall interface, events that do not use this interface cannot be monitored or modified by HyDE programs. For example, HyDE could not easily implement a shared clipboard between a guest and host as no syscalls are involved in this interaction.

Beyond these limitations, we see several areas for future work. HyDE performance could potentially be enhanced through optimization of the syscall event detection logic. With minor engineering effort, HyDE could be extended to support additional operating systems such as Windows and macOS. Additionally, HyDE could be extended to support analyzing and modifying guest state at non-syscall events by dynamically patching other logic to trigger placeholder syscall instructions.

9. Related Work

HyDE advances the state of the art by enabling system-wide analysis and control of unmodified guests without the understanding guest internals. However, we are far from the first to explore the challenges of monitoring and controlling guest systems.

Garfinkel et al. [3] analyze the behavior of an emulated guest from a VMM using VMI to identify intrusions. VMI-based approaches require bridging the semantic gap [25] which typically requires a deep understanding of the frequently-changing internals of a guest kernel [26].

With *IntroVirt*, Joshi et al. [27] identify that VMI can be supplemented by executing in-guest logic (such as syscalls) to sidestep parts of the semantic gap. IntroVirt adds breakpoints into guest code at events of interest and executes custom "predicate" programs when the breakpoints are triggered. A snapshot/restore system is used to revert guest state after each predicate runs. Though these predicates are conceptually similar to HyDE programs, they are tightly coupled with low-level guest internals: binaries with debugging symbols are required, and guest functionality is invoked with direct function calls to guest code.

System call tracing is a common technique for malware analysis and detection. Bayer et al. [28] and Jiang et al. [15] demonstrate that syscall traces can be collected from an emulated guest without the need for VMI by modifying QEMU. Dinaburg et al. [13] extend this approach to support virtualized guests.

Gu et al. [29], Carbone et al [30], and Fu et al. [31] all demonstrate how an isolated, out-of-guest process can run logic within a guest system using process implanting, function call injection, and system call forwarding, respectively. While these approaches enable introspection and modification of the guest system, they differ from HyDE as they cannot react to specific guest events and lack the ability to interpose on *all* guest processes. For example, these approaches would be unable to enforce a system-wide security policy like the one described in Fig. 2.

10. Conclusion

We have presented HyDE, a novel approach for programming guest processes from a VMM by examining, modifying, and injecting syscalls. We described the design of HyDE, detailed our implementation, and presented 10 example programs built atop HyDE.

We thoroughly test HyDE by validating that syscall injection can be state-preserving and then testing the reliability and performance of guests when running with each example HyDE program. Our results show that HyDE programs can accomplish real-world user goals from a VMM without sacrificing guest reliability. Furthermore, our results show the HyDE performance overhead to be negligible for one-off use cases and, for some users, a worthy trade-off for persistent applications. Through this work, we demonstrated the reliability and utility of programming guest processes with syscall injection. To enable others to build off this framework and deploy this approach to additional problems, we have released the code for our HyDE implementation and example HyDE programs.

Availability

Our HyDE implementation, software development kit, and example programs are available at https://github.com/AndrewFasano/hyde.

Acknowledgments

The authors would like to thank Jacques Becker, Luke Craig, Jordan McLeod, and the anonymous reviewers for their feedback on this work. OpenAI's GPT 3.5 assisted with the development of the HyDE SDK, specifically the `yield_syscall` macro.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited. This material is based upon work supported by the Dept of the Army under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Dept of the Army. Delivered to the U.S. Government with Unlimited Rights, as defined in DFARS Part 252.227-7013 or 7014 (Feb 2014). Notwithstanding any copyright notice, U.S. Government rights in this work are defined by DFARS 252.227-7013 or DFARS 252.227-7014 as detailed above. Use of this work other than as specifically authorized by the U.S. Government may violate any copyrights that exist in this work.

References

[1] A. V. Aho and J. D. Ullman, *Foundations of computer science*. Computer Science Press, Inc., 1992.

[2] S. Bhardwaj, L. Jain, and S. Jain, "Cloud computing: A study of infrastructure as a service (iaas)," *International Journal of engineering and information Technology*, vol. 2, no. 1, pp. 60–63, 2010.

[3] T. Garfinkel, M. Rosenblum *et al.*, "A virtual machine introspection based architecture for intrusion detection." in *Ndss*, vol. 3, no. 2003. Citeseer, 2003, pp. 191–206.

[4] J. Hwang, S. Zeng, F. y Wu, and T. Wood, "A component-based performance comparison of four hypervisors," in *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*. IEEE, 2013, pp. 269–276.

[5] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, 1974.

[6] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux symposium*, vol. 1, no. 8. Dttawa, Dntorio, Canada, 2007, pp. 225–230.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS operating systems review*, vol. 37, no. 5, pp. 164–177, 2003.

[8] J. Sugerman, G. Venkitachalam, and B.-H. Lim, "Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor." in *USENIX Annual Technical Conference, General Track*, 2001, pp. 1–14.

[9] J. W. Rittinghouse and J. F. Ransome, *Cloud computing: implemntation, management, and security*. CRC press, 2016.

[10] S. Wu, L. Deng, H. Jin, X. Shi, Y. Zhao, W. Gao, J. Zhang, and J. Peng, "Virtual machine management based on agent service," in *2010 International Conference on Parallel and Distributed Computing, Applications and Technologies*. IEEE, 2010, pp. 199–204.

[11] F. Bellard, "Qemu, a fast and portable dynamic translator," in *USENIX Annual Technical Conference, FREENIX Track*, 2005.

[12] J. Pfoh, C. Schneider, and C. Eckert, "Nitro: Hardware-based system call tracing for virtual machines," in *Advances in Information and Computer Security: 6th International Workshop, IWSEC 2011, Tokyo, Japan, November 8-10, 2011. Proceedings 6*. Springer, 2011, pp. 96–112.

[13] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in *Proceedings of the 15th ACM conference on Computer and communications security*, 2008, pp. 51–62.

[14] B. Li, J. Li, T. Wo, C. Hu, and L. Zhong, "A vmm-based system call interposition framework for program monitoring," in *2010 IEEE 16th International Conference on Parallel and Distributed Systems*. IEEE, 2010, pp. 706–711.

[15] X. Jiang and X. Wang, ""out-of-the-box" monitoring of vm-based high-interaction honeypots," in *Recent Advances in Intrusion Detection: 10th International Symposium, RAID 2007, Gold Goast, Australia, September 5-7, 2007. Proceedings 10*. Springer, 2007, pp. 198–218.

[16] D. Wheeler, "Sloccount," *http://www. dwheeler. com/sloccount/*, 2001.

[17] T. U. S. D. of Commerce, "The minimum elements for a software bill of materials (SBOM)," https://www.ntia.doc.gov/files/ntia/publications/sbom_minimum_elements_report.pdf, 2021.

[18] J. Horejsi and D. Fiser, "Analysis of kinsing malware's use of rootkit," https://www.trendmicro.com/en_us/research/20/k/analysis-of-kinsing-malwares-use-of-rootkit.html, 2020.

[19] J. Junnila, "Effectiveness of linux rootkit detection tools," Master's thesis, University of Oulu, 2020.

[20] "GNU Coreutils," https://www.gnu.org/software/coreutils/, GNU Project, 2020.

[21] L. W. McVoy, C. Staelin *et al.*, "lmbench: Portable tools for performance analysis." in *USENIX annual technical conference*. San Diego, CA, USA, 1996, pp. 279–294.

[22] T. Palit, Y. Shen, and M. Ferdman, "Demystifying cloud benchmarking," in *2016 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 2016, pp. 122–132.

[23] S. Rostedt, "Ftrace linux kernel tracing," in *Linux Conference Japan*, 2010.

[24] B. Gregg, "Linux performance analysis new tools and old secrets," in *Usenix Lisa 2014 conference*, 2014.

[25] P. M. Chen and B. D. Noble, "When virtual is better than real [operating system relocation to virtual machines]," in *Proceedings eighth workshop on hot topics in operating systems*. IEEE, 2001, pp. 133–138.

[26] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion, "Sok: Introspections on trust and the semantic gap," in *2014 IEEE symposium on security and privacy*. IEEE, 2014, pp. 605–620.

[27] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen, "Detecting past and present intrusions through vulnerability-specific predicates," *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5, pp. 91–104, 2005.

[28] U. Bayer, C. Kruegel, and E. Kirda, *TTAnalyze: A tool for analyzing malware*. na, 2006.

[29] Z. Gu, Z. Deng, D. Xu, and X. Jiang, "Process implanting: A new active introspection framework for virtualization," in *2011 IEEE 30th International Symposium on Reliable Distributed Systems*. IEEE, 2011, pp. 147–156.

[30] M. Carbone, M. Conover, B. Montague, and W. Lee, "Secure and robust monitoring of virtual machines through guest-assisted introspection," in *Research in Attacks, Intrusions, and Defenses: 15th International Symposium, RAID 2012, Amsterdam, The Netherlands, September 12-14, 2012. Proceedings 15*. Springer, 2012, pp. 22–41.

[31] Y. Fu, J. Zeng, and Z. Lin, "HYPERSHELL: A practical hypervisor layer guest OS shell for automated in-vm management," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014, pp. 85–96.

Appendix A. Correctness Testing of Example HyDE Programs

We programmatically evaluate the following questions to test the correctness of our HyDE programs. Prior to running any experiments, our infrastructure first runs a guest without the HyDE program loaded and asserts that the correctness test fails. After this, the infrastructure runs experiments with the HyDE program loaded and reports the results of the correctness tests.

A.1. One-Off Programs

GetSysInfo: Does the number of processes reported during the test roughly align with the number of processes running in the guest after the test?
HyperPS: Does the output of running *ps* in the guest after the test largely match the *HyperPS* generated output?
PwReset: After the test, can the root user log in with the newly set password?

A.2. Persistent Programs

2faRoot: When a guest tries to escalate privileges, does an out-of-band user have to approve the escalation? If the request is approved, can the user escalate privileges? If the request is denied, does the user fail to escalate privileges?
DSBOM: Are accessed files in the guest logged to the host? Do the hashes of these files match expected values?
EnvAdder: Do newly launched guest processes see the added environment variable?
FileAccessLog Is a randomly-accessed filename in the guest logged to the host?
NoRootSock: Does the command *sudo nc -lp 80* raise an error? Can a custom C webserver that *bind*s to port 80 and drops privileges before *listen*ing to the socket successfully listen for traffic?
SecretFile: Can only some guest processes read the secret file? Does it contain the correct contents?
UntrustedRoot: Are root users unable to *strace* a process?

Appendix B. Emulation-Based HyDE for Cross-Architecture Guests

Beyond the HyDE implementation described in this work, we have previously created an implementation of HyDE atop the QEMU emulator with support for emulated x86, arm, and mips (big and little endian) guests. This implementation allowed HyDE programs to be developed as Python 3 scripts using a similar asynchronous programming model as the HyDE implementation described in this work. The HyDE-SDK provided with that implementation included an architecture-neutral interface for creating abstract syscall objects, allowing users to examine and create syscalls without having to know the details of the underlying architecture.

We developed three HyDE programs atop this implementation. The first is the *EnvAdder* program previously described. The second, enabled remote user-level interactive debugging of an target guest processes process from outside a guest, as if the process were running under `gdbserver` . After launching this HyDE program, a user could launch `gdb` , connect to the HyDE program's listening port, and interactively debug the target process. The third program would dynamically reconfigure guest network sockets to transparently use the AF_VSOCK address family instead of AF_INET. We evaluated the reliability, portability, and performance of these three programs across x86_64, ARM, MIPS big endian and MIPS little endian guests running Linux and FreeBSD kernels. In total, we evaluated 39 distinct OSes and kernel versions across these architectures and found the HyDE programs to behave properly in all tests. After creating this initial prototype and discovering the value of HyDE, we elected to reimplement HyDE with support for virtualized guests as virtualization is widely used.

Appendix C. Coreutils Test Suite Variance

When conducting our evaluations with Coreutils, by running `make check SUBDIRS=. -k -j $(nproc)` as a non root user, we find different numbers of test are selected to run across our different guest systems. This is due to the fact that individual tests have dependencies on system configuration and the presence of various files and utilities. Within a single guest, we also observed some variance in selected tests, even when running with stock KVM. For our two Ubuntu guests, we see two tests nondeterministically selected for inclusion in the test suite in the baseline configuration. For Centos8 we see four tests nondeterministically selected. One HyDE program, *UntrustedRoot* does alter the number of tests coreutils selects to run. This occurs because some of the coreutils tests are only run if the guest is able to take some actions that *UntrustedRoot* blocks. As such, coreutils selects approximately ten fewer tests to run. We present the average number of tests and standard deviation in Table 2.

TABLE 2. MEAN NUMBER OF TESTS SELECTED TO RUN BY THE COREUTILS TEST SUITE IN EACH EXPERIMENT.

	Ubuntu 23.04	Ubuntu 22.04	CentOS 8
Baseline KVM	517.60 ± 0.70	521.20 ± 0.42	493.10 ± 1.37
HyDE VMM	517.60 ± 0.70	521.50 ± 0.53	493.00 ± 1.63
2faRoot	517.40 ± 0.52	521.30 ± 0.48	493.00 ± 1.05
DSBOM	517.60 ± 0.70	521.20 ± 0.42	492.90 ± 0.88
EnvAdder	517.10 ± 0.32	521.60 ± 0.70	493.00 ± 0.82
FileAccessLog	517.40 ± 0.52	521.40 ± 0.52	493.00 ± 1.05
GetSysinfo	517.60 ± 0.70	521.20 ± 0.42	492.90 ± 0.99
NoRootSocks	517.20 ± 0.42	521.60 ± 0.70	492.60 ± 0.84
HyperPS	517.20 ± 0.42	521.70 ± 0.82	492.60 ± 1.07
PwReset	517.20 ± 0.79	521.70 ± 0.82	492.80 ± 1.03
SecretFile	517.30 ± 0.67	521.60 ± 0.70	492.50 ± 0.71
UntrustedRoot	510.80 ± 0.42	511.20 ± 0.42	484.10 ± 1.10
Overall	516.83 ± 1.91	520.6 ± 2.97	492.13 ± 2.53

Appendix D.
Commercial Cloud Provider Documentation

D.1. Password Reset

Amazon AWS EC2, DigitalOcean, and Linode all provide password reset mechanisms for their virtualized guests, though their techniques may fail depending on guest configuration. Documentation for these features is available at

- https://docs.aws.amazon.com/systems-manager/latest/userguide/automation-ec2reset.html
- https://docs.digitalocean.com/support/how-do-i-reset-my-droplets-root-password/
- https://www.linode.com/docs/products/compute/compute-instances/guides/reset-root-password/

D.2. Security Monitoring

Amazon and Google both provide agents that can be installed within a guest to provide security features. Amazon's guest agent is called Inspector: https://aws.amazon.com/inspector/ while Google provides "security software agents": https://cloud.google.com/marketplace/docs/deploy-security-software-agents.