Decoupling dynamic program analysis from execution in virtual environments

Jim Chow Tal Garfinkel Peter M. Chen
VMware

Abstract

Analyzing the behavior of running programs has a wide variety of compelling applications, from intrusion detection and prevention to bug discovery. Unfortunately, the high runtime overheads imposed by complex analysis techniques makes their deployment impractical in most settings. We present a virtual machine based architecture called Aftersight ameliorates this, providing a flexible and practical way to run heavyweight analyses on production workloads.

Aftersight *decouples* analysis from normal execution by logging nondeterministic VM inputs and replaying them on a separate analysis platform. VM output can be gated on the results of an analysis for intrusion prevention or analysis can run at its own pace for intrusion detection and best effort prevention. Logs can also be stored for later analysis offline for bug finding or forensics, allowing analyses that would otherwise be unusable to be applied ubiquitously. In all cases, multiple analyses can be run in parallel, added on demand, and are guaranteed not to interfere with the running workload.

We present our experience implementing Aftersight as part of the VMware virtual machine platform and using it to develop a realtime intrusion detection and prevention system, as well as an an offline system for bug detection, which we used to detect numerous novel and serious bugs in VMware ESX Server, Linux, and Windows applications.

1 Introduction

Dynamic program instrumentation and analysis enables many applications including intrusion detection and prevention [18], bug discovery [11, 26, 24] and profiling [10, 22]. Unfortunately, because these analyses are executed inline with program execution, they can substantially impact system performance, greatly reducing their utility. For example, analyses commonly used for detecting buffer overflows or use of undefined memory routinely incur overheads on the order of 10-40x [18, 26], rendering many production workloads unusable. In non-production settings, such as program development or quality assurance, this overhead may dissuade use in longer, more realistic tests. Further, the performance perturbations introduced by these analyses can lead to Heisenberg effects, where the phenomena under observation is changed or lost due to the measurement itself [25].

We describe a system called *Aftersight* that overcomes these limitations via an approach called *decoupled analysis*. Decoupled analysis moves analysis off the computer that is executing the main workload by separating execution and analysis into two tasks: *recording*, where system execution is recorded in full with minimal interference, and *analysis*, where the log of the execution is replayed and analyzed.

Aftersight is able to record program execution efficiently using virtual machine recording and replay [4, 9, 35]. This technique makes it possible to precisely reconstruct the entire sequence of instructions executed by a virtual machine, while adding only a few percent overhead to the original run [9, 35]. Further, as recording is done at the virtual machine monitor(VMM) level, Aftersight can be used to analyze arbitrary applications and operating systems, without any additional support from operating systems, applications, compilers, etc.

Aftersight supports three usage models: synchronous safety, best-effort safety, and offline analysis. First, for situations where timely analysis results are critical (e.g., intrusion detection and prevention), Aftersight executes the analysis in parallel with the workload, with the output of the workload synchronized with the analysis. This provides synchronous safety that is equivalent to running the analysis inline with the workload. Second, for situations that can tolerate some lag between the analysis and the workload, Aftersight runs the analysis in parallel with the workload, with no synchronization between the output of the workload and the analysis. This best-effort safety allows the workload to run without being slowed by the analysis. Often analyses whose performance impact would be prohibitive if done inline can run with surprisingly minimal lag if run in parallel. Third, Aftersight can run analyses offline for situations where analyses are not known beforehand or are not time critical, such as when debugging.

Aftersight is a general-purpose analysis framework. Any analysis that can run in the critical path of execution can run in Aftersight, as long as that analysis does not change the execution (this would break the determinism that Aftersight's replay mechanism relies upon). Also, Aftersight makes the entire system state at each instruction boundary available for analyses, providing greater generality than approaches based on sampling. Further, logs originating from the VMM can be replayed and analyzed in different execution environments(e.g., a simulator or VMM). This flexibility greatly eases program instrumentation and enables a variety of optimizations.

We have implemented an Aftersight prototype on the *x*86 architecture, building on the record and replay capability of VMware Workstation. Our framework enables replay on the QEMU whole-system emulator, which supports easy instrumentation during replay and analysis. With this framework, we have implemented an online security analysis that can be used to detect buffer overflow attacks on running systems. We also implemented an analysis that can perform checks for memory safety and heap overflows, and we used this analysis to discover several new and serious bugs in VMware ESX Server, Linux, and Windows applications.

2 The case for decoupled analysis

Aftersight improves dynamic analysis by decoupling the analysis from the main workload, while still providing the analysis with the identical, complete sequence of states from the main workload. This combination of decoupling and reproducibility improves dynamic analysis in the following ways.

First, Aftersight allows analyses to be added to a running system without fear of breaking the main workload. Because Aftersight runs analyses on a separate virtual machine from the main workload, new analyses can be added without changing the running application, operating system, or virtual machine monitor of the main workload.

Second, Aftersight offers users several choices along the safety/performance spectrum. Users who can tolerate some lag between the analysis and the workload can improve the performance of the workload and still get best-effort safety or offline analysis, while users who require synchronous safety can synchronize the output of the workload with the analysis.

Third, with best-effort safety or offline analysis, Aftersight can improve latency for the main workload by moving the work of analysis off the critical path. Because analyses no longer slow the primary system's responsiveness, heavyweight analyses can now be run on realistic workloads and production systems without fear of perturbing or unduly slowing down those workloads. For example, system administrators can use intensive checks for data consistency, taint propagation, and virus scanning on their production systems. Developers can run intensive analyses for memory safety and invariant checking as part of their normal debugging, or as additional offline checks that augment testing that must already be performed in a quality-assurance department. As an extreme illustration of the type of heavyweight analysis enabled by Aftersight, computer architects can capture the execution of a production system with little overhead, then analyze the captured instruction stream on a timing-accurate, circuit-level simulator. Even when providing synchronous safety, Aftersight can sometimes improve performance compared to running the analysis inline by leveraging the parallel execution of the workload and the analysis.

Fourth, Aftersight increases the parallelism available in the system by providing new ways to use spare cores. Aftersight can run an analysis in parallel with the main workload, and it can run multiple analyses in parallel with each other.

Fifth, Aftersight makes it feasible to run multiple analyses for the exact same workload. Without Aftersight, the typical way to run multiple analyses is to conduct a separate run per analysis, but this suffers from the likelihood of divergent runs and inconsistent analyses. Aftersight, in contrast, guarantees that all analyses operate on the same execution. In addition, each analysis takes place independently, so programmers need not worry about unforeseen interactions between the analyses. Nor must they worry about perturbing the source workload with their analysis. Aftersight allows the number of simultaneous analyses to scale with the number of spare processors in a system, all while not affecting the performance of the primary system.

Sixth, Aftersight makes it possible to conduct an analysis that was not foreseen during the original run. This *ex post facto* style of analysis is particularly powerful when it is difficult to anticipate exactly what must be analyzed. For example, analyzing computer intrusions invariably requires one to examine in detail a scenario that was not foreseen (else, one would have prevented the intrusion). Debugging performance or configuration problems leads to a similar need for conducting unforeseen analysis. Aftersight allows the user to iteratively develop and run new analyses, all on the same exact execution.

Finally, by decoupling analysis from the main execution, Aftersight allows the analysis and execution components to be individually optimized to their in-

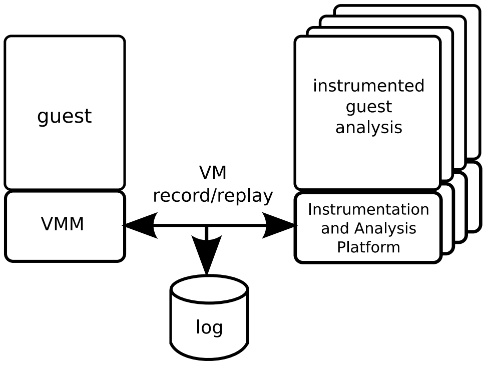

Figure 1: Overview of current system.

tended function. The main workload execution can be performed on a platform optimized for realtime performance and responsiveness (such as a commercial VMM), while analysis can be delegated to a platform optimized for ease of instrumentation (such as an extensible simulator).

3 Architectural overview

Our current Aftersight system targets the *x*86 architecture and has three main components: the virtual machine monitor (VMM), deterministic VM record/replay, and an analysis and instrumentation framework. For our prototype, each of these pieces builds on functionality of existing off-the-shelf components. In this section we examine aspects of these components that are relevant for decoupled analysis. In Section 5 we look at how they can be modified and integrated to facilitate decoupled analysis.

3.1 VMM

A VMM provides an environment for running arbitrary guest operating systems and applications in a software abstraction of the hardware [23]. No modifications to a standard VMM are required to support decoupled analysis, except for the need to support replay. Aftersight uses VMware Workstation, a highly optimized production *x*86 VMM. VMware Workstation uses a hosted architecture [28], i.e. it uses a *host* operating system to access physical devices like disk or network.

As Aftersight is based on a virtual machine platform, it inherits a variety of useful and desirable traits. First, any individual process in the guest VM as well as the guest OS kernel itself can be a target of Aftersight instrumentation and analysis. Furthermore, a range of target systems are supported without extra work: a single implementation works across OSes, processes, languages, etc.

Next, virtualization is becoming increasingly ubiquitous in a wide range of computing environments. Because Aftersight can be provided as a service of the virtual infrastructure with nominal overhead, there is a relatively easy path to adoption. With such a primitive in place, the deployment of new monitoring and analysis tools can be a continuous, normal part of the execution of guests.

Finally, operating at the VMM level gives Aftersight visibility at all layers of the software stack. It can be used to analyze operating systems, applications, and interactions across components. This generality is critical for applications ranging from performance analysis, to tracking the flow of sensitive or potentially malicious data in a system.

3.2 Deterministic VM record and replay

Aftersight builds upon the replay facilities in VMware Workstation [1]. A deterministic VM record/replay system records enough information about a running workload to reproduce its exact instruction sequence. To support replay, a VM must record and replay all inputs to the CPU that are not included in the state of the guest memory, registers, or disk. This includes reads from external devices, such as the network, keyboard, or timer, and asynchronous events such as interrupts. Recording these nondeterministic inputs enable VM replay to recreate the whole instruction stream [4, 9, 35]. As with other software-based replay systems [9], VMware Workstation is not able to replay virtual multiprocessors.

VM replay systems are highly efficient in time and space overhead [9, 35]. This efficiency comes about because nearly all instructions produce the same result given the same inputs, and most instructions use only the results of previous instructions as their inputs. Because of this domino effect, a long sequence of instructions can be exactly reproduced while only supplying a few values that come “from outside” the system.

A study [35] of VMware’s then-current replay implementation showed performance overheads for SPEC benchmarks as low as 0.7% and an average of 5% [35]. Another replay implementation for the *x*86 [9] reported similar overheads. Overheads will generally be workload dependent, however. Worst-case performance observed in [35] reached 31% and 2.6x for some workloads. However, many of the chief bottlenecks were not fundamental [35], and subsequent improvements have lowered these overheads.

Trading the overhead of analysis for the overhead of VM replay is a compelling exchange for many heavyweight analyses. However, even for lightweight analyses, the ability to run multiple or *ex post facto* analyses still provides reason to use decoupled analysis.

While Aftersight uses VM replay, replay can also be implemented at many levels besides the VM-level, such as the OS process-level [27], the JVM-level [8], or the

[1] First released in VMware Workstation 6, where it was an experimental feature.

disk level [32]. VM replay stands out for a couple reasons.

First, a single VM replay implementation enables one to replay the entire state of all software on that hardware platform regardless of operating system, language or runtime environment. In contrast, other types of replay can only work for a small subset of available software as they are heavily dependent on the particular language and operating system they are designed for. Reimplementing replay for each OS or language variant would be a herculean task.

Second, a VM replay system often has lower overhead than higher-level recording. For example, one could replay the file system at the system-call level, but this would require all file system calls to be recorded, including every `read` of every file. However, a VM-based system need only record nondeterministic VM inputs, and this frees it from recording reads from the file cache or disk. Similarly, a process-level record/replay system must record reads from IPC pipes, files, and all system calls that return data, while these can all be ignored by a VM-based solution.

3.3 Analysis framework

Aftersight does instrumentation and analysis dynamically during replayed execution. Normally in a VM record/replay system, the same VMM is used during both recording and replaying. A key property of Aftersight is its ability to support *heterogenous replay*, i.e. the ability to use one platform to execute and record a workload and a different platform to replay and analyze, with each platform tuned for its particular purpose.

The Aftersight prototype relies on a VMM for execution and recording, whereas replay and analysis can be done in a VMM or a simulator. A VMM is an excellent platform for recording, because it is optimized to minimize recording overhead to support production environments. However, platforms such as software simulators are often better suited to supporting general-purpose analysis.

For example, many VMM environments don't provide a simple, low overhead way to instrument every memory access. This can be implemented on top of page protections and faulting on every memory access. However, a software simulator can often accomplish this task faster and with less effort than a VMM will natively.

Analysis environments Dynamic instrumentation can be implemented in many ways. Most simply, we can build ad-hoc hooks into our replaying environment that supply callbacks when events of interest happen.

In our Aftersight prototype, we implement dynamic instrumentation through dynamic binary translation (BT). BT is the technique of dynamically translating a set of instructions into an alternate set of instructions on-the-fly, which are then executed. Techniques such as caching translations [33] can be used to make this process very efficient. Affecting what translations are produced allows one to very flexibly instrument a running program.

Our prototype offers two BT environments to analysis applications: one is based on VMware Workstation, the other on QEMU [3], which is an open-source *x*86 simulator. Both offer the ability to run code using BT alone, or in some combination with native execution [1]. However, each has its own strengths and limitations.

VMware's BT is extremely fast, but it is optimized for performance rather than extensibility. For example, VMware's BT does not support an extensible intermediate representation (IR). An extensible IR is commonly used in general purpose BT systems [17, 3] to abstract the *x*86's CISC-style instructions into a more instrumentation-friendly RISC-style format. However, these additional translation costs make little sense given VMware's specialized use of BT. Also, for efficiency reasons, VMware BT runs in ring 0, and in an environment where dynamic memory allocation is heavily constrained. Developing general-purpose analyses under these constraints is quite burdensome, and the resulting analyses may even be slower from having to work with limited memory.

In contrast, QEMU is not nearly as fast as VMware Workstation, but it is much more flexible: it provides an extensible IR and runs as a regular user-mode process, which means normal program facilities like `malloc`, `gdb`, etc. are available. In converting it to enable replay, we stripped out much of its now unnecessary functionality, to the point where it is little more than a simple CPU simulator. The virtual device model, including the disk, network, the chipset, and the local APIC, have all been removed. All that remains are the components needed to deal with instruction execution and memory access.

Of course, because analysis is decoupled, other special-purpose analysis environments could be built to better suit the needs of particular analyses if desired.

4 Online analysis

In two of Aftersight's three usage models, synchronous safety and best-effort safety, the analysis runs in parallel with the workload. Aftersight makes it easy to simultaneously record and analyze a workload. In our prototype, recording generates a replay log on disk. Analysis VMs can run on separate cores and process the log as it is being generated by the primary VM. Analysis can even take place across multiple machines by reading the log file over the network, since network bandwidth is more than adequate for most workloads. Log sizes are often quite modest [9, 35]. For example, Xu et al. [35] notes that only 776 KB of compressed log space was necessary to record an entire Windows XP bootup-shutdown sequence.

This section describes how Aftersight synchronizes the main workload with the analysis when the two are running in parallel, and how running the analysis in parallel with the workload can speed up the analysis.

4.1 Synchronization

When running in simultaneous record and analysis mode, analysis results may affect the operation of the primary VM (e.g., a security check may detect an intrusion and halt the system). When this feedback is needed, Aftersight can take one of two strategies to synchronize the execution of the primary and analysis VMs.

The need for synchronization arises because the primary VM executes ahead of the analysis VM. The portion of the primary VM's execution that has not yet been run on the analysis VM is *speculative*. This speculative portion will usually be committed by the analysis VM as checks complete. In the rare case when checks fail, the speculative portion of execution differs from what would have been executed with inline analysis.

The first method for synchronizing provides synchronous safety, which is equivalent to running the analysis inline with the workload. To provide this guarantee, Aftersight defers the output of the primary VM (e.g., network packets) while they are speculative, i.e., until the analysis reaches the same point in execution. Deferring outputs while they are speculative ensures that the released outputs of the primary VM are identical to those of a system with inline checks, even though the internal state of the primary VM may differ from a system with inline checks [19].

In addition to synchronizing the primary's output with the analysis VM, we could also limit how far the primary is allowed to run ahead of the analysis VM. Limiting the lag between primary and backup limits the amount of time that the primary's outputs are deferred, which in turn limits the amount of timing perturbation the primary VM may observe (e.g., when it measures the round-trip time of a network).

Deferring output in the above manner provides the same safety guarantee as if the analysis were running inline with the workload. However, it may hurt performance by blocking the output of the primary VM.

A different point in the safety/performance spectrum optimizes performance but relaxes the safety guarantee by giving lazy feedback to the primary VM. In this case, the main workload executes at full speed and is not slowed by the work of analysis. Rather, analysis results are fed back to the main workload as they become available. This usage model is useful when the analysis is too heavyweight to run with stronger safety guarantees, or when the analysis does not require such guarantees, as in profiling or debugging.

4.2 Accelerating analysis

The analysis VM in Aftersight executes the same instructions as the primary VM, and it also does the work of analysis. Because the analysis VM is doing more work than the primary VM, it can easily become a bottleneck, especially when providing synchronous safety. This section describes several ways to improve the performance of the analysis VM, to allow it to better keep up with the primary VM.

First, a surprising amount of performance can be won from a basic aspect of replayed VM execution: interrupt delivery is immediate. *x*86 operating systems use the `hlt` instruction to wait for interrupts; this saves power compared to idle spinning. During analysis, `hlt` time passes instantaneously. One `hlt` invocation waiting for a 10ms timer interrupt can consume equal time to tens of millions of instructions on modern 1+GHz processors. Section 6.3 provides more detail on the boost this can have on performance.

Second, device I/O can be accelerated during replay. For example, network writes need not be sent, and network reads can use data from the replay log. This frees guests from waiting for network round-trip times, especially because disk throughput is often greater than end-to-end network throughput. Disk reads can similarly be satisfied from the replay log rather than from the analysis VM's disk, and this can accelerate the analysis VM because the replay log is always read sequentially. This optimization can also free the analysis VM from executing disk writes during replay, which frees up physical disk bandwidth and allows write completion interrupts to be delivered as soon as the guest arrives at an appropriate spot to receive them. Disk reads done by the primary VM may also prefetch data and thereby accelerate subsequent reads by the analysis VM [5].

Third, a number of opportunities allow Aftersight to *memoize* operations that happen during record that don't need to be fully replayed. An example of this is exception checking.

There are many times where the *x*86 needs to check for exceptional conditions. Although these checks rarely raise exceptions, executing them adds considerable overhead in our CPU simulator. Segment limit checks are an example: every memory reference or instruction fetch must be checked that it is within bounds for an appropriate segment [2].

[2] These add enough overhead that QEMU completely ignores the behavior. This turns out to work for many workloads, but not all. Playing fast and loose with the specification in this way inevitably causes failures—non-executable stacks are popularly implemented with segment limits in major *x*86 Unix derivatives where non-executable page protections are unavailable (which is true for all non-PAE kernels), but QEMU makes them behave incorrectly.

Decoupled analysis allows one to reduce the overhead of exception checking on the analysis VM by leveraging the exception checking that has already occurred on the main VM. The time and location in the instruction stream of any exceptions are recorded by the main VM, and these exceptions are delivered during replay just like asynchronous replay events. This strategy frees the analysis VM from the overhead of explicitly checking for exceptions during replay. Memoizing these checks makes the CPU simulator faster and less complex, while still guaranteeing proper replay of a workload that contains violations of the checks.

There are many *x*86 checks that can be memoized, although we have not yet implemented this optimization in Aftersight: debug exceptions, control transfer checks for segment changes, the alignment check (which when enabled, ensures all memory accesses are performed through pointers aligned to appropriate boundaries), and others.

5 Implementation and integration

While Aftersight builds on existing components, leveraging these for decoupled analysis poses a variety of challenges. This section discusses how to adapt a simulation environment to replay VMM logs and the challenges posed by the heterogeneous combination of record and replay components.

Aftersight uses different platforms for recording and analysis. For recording, Aftersight uses VMware Workstation, which is designed to minimize the time and space overhead of recording. For analysis, Aftersight can use VMware Workstation or QEMU. Simple analyses can be conducted by modifying VMware Workstation's BT, while more general analyses are easiest to implement in QEMU, which is designed for flexibility rather than pure speed.

For replay and analysis, compatibility is an issue for both platforms. When VMware Workstation replays a log, no compatibility issues arise with devices, chipset, etc. because the emulation code is identical. However, because it relies directly on the hardware for CPU emulation, replay is generally infeasible if the processor is significantly different (e.g., attempting to replay a log from an Intel CPU on an AMD platform). In contrast, with a CPU simulator like QEMU we can easily support a wide range of CPU families on a single hardware platform. However, QEMU does not have the same device models as the recording platform. In this next section, we look at how Aftersight bridges the compatibility gap between the VMware Workstation recording and QEMU in two areas: I/O device emulation and hardware performance counters.

5.1 Device emulation

The first gap between our recording platform (VMware Workstation) and one of our analysis platforms (QEMU) is device emulation. QEMU emulates different I/O devices than VMware, which prevents QEMU from directly consuming the log recorded by VMware.

To understand the problem and the solution we adopted, it is helpful to consider two different methods for recording device interactions. The first method is to record all outputs from an emulated device to the CPU. During replay, the recorded values would be resupplied to the CPU (presumably the guest OS device drivers). This method is ideal for compatibility between the recording and analysis platform because no device emulation is needed during replay.

However, VMware Workstation and other VM replay systems [9] use a second method to record and replay device interactions. Instead of recording the output from the emulated devices, they record the nondeterministic, external inputs to those devices. During replay, these recorded inputs are redelivered to the devices, and these allow the emulated devices to be deterministically replayed along with the CPU.

VM replay systems use this second method for two reasons. A main reason is that the second method allows a replaying session to "go live"—to stop replaying and start responding to new input—at any point while replaying. In contrast, recording and replaying the outputs of the emulated device without replaying the emulated device itself means that the emulated device is not available to go live. Another reason is that it can drastically reduce the amount of data that must be recorded. For example, to replay a disk read operation, the first method must record the actual data being read from the emulated disk, while the second method need only record the nondeterministic inputs to the disk (note that the inputs from the CPU to the disk are deterministic and need not be recorded).

Unfortunately, recording only the nondeterministic inputs to the device leads to a compatibility problem during analysis. Whereas the VM recording system assumes that the replaying system can replay the emulated device, QEMU and other flexible analysis systems usually will not emulate the exact same devices used during recording.

Aftersight bridges the compatibility gap between recording and analysis for devices by adding a *relogging* step to replay. We modified VMware's replay system to record a new log during replay, which contains all outputs from the emulated device to the CPU, including responses to I/O requests, interrupt delivery, and effects on memory. This log is equivalent to one generated by the first method of replaying devices and has the same compatibility advantages, i.e. the analysis system needs no device emulation during replay.

While our modified VMware VMM supports relogging, none of our modifications to support relogging impact the *record* side operation of the VMM, since relogging is only active during replay.

5.2 Hardware performance counters

The second gap between our recording platform (VMware Workstation) and one of our analysis platforms (QEMU) is hardware performance counters. VM replay implementations will normally use hardware counters to determine when a nondeterministic event happens during recording, as well as to trigger that event during replay [4, 9]. These counters record aspects of the dynamic instruction stream that help to uniquely position an event in time such as the instruction count [4], or the number of branches executed [9].

Instructions added dynamically by BT, as well as by analysis instrumentation, disrupt counts kept by the hardware by adding dynamic instructions in an unpredictable manner. This makes hardware counters difficult to use directly by our CPU simulator.

Instead, our CPU simulator emulates the accounting provided by the hardware counters in the translations it emits. These translations include a small amount of code to update counts and dispatch to an event handler when it is time to deliver an asynchronous nondeterministic event (such as an interrupt or DMA).

QEMU doesn't normally allow interrupt delivery within a basic block [2] of instructions. Instead, these events are delayed until the current basic block completes. The VMware recording system contains no such artificial restriction, so we needed to remove this restriction of QEMU to replay VMware's log. When our stripped-down QEMU reaches a basic block containing a replay event, it will emit new translations for the block. The block is split into two halves: the block of instructions before the replay event, and the block after. Checks between basic blocks will determine that the BT system can deliver the event.

6 Evaluation

Aftersight makes it possible to run heavyweight analyses on realistic workloads with several options along the safety/performance spectrum. In this section, we evaluate the performance of Aftersight under three usage models. We first show how Aftersight can provide synchronous safety with slightly higher performance than a system using inline analysis. Next, we show how Aftersight with best-effort safety makes it possible to run heavyweight analyses in parallel with the main workload and how the techniques described in Section 4.2 allow heavyweight analyses to keep up with the main workload. Last, we demonstrate the utility of enabling heavyweight analyses that are built and applied after the main workload completes.

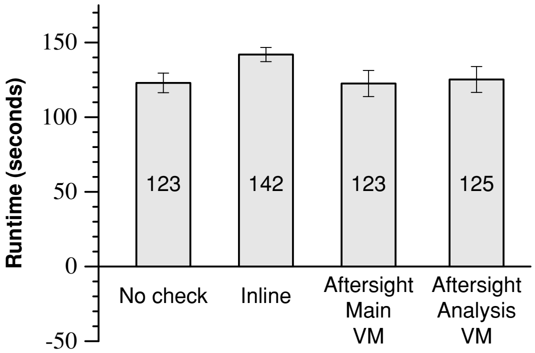

Figure 2: Aftersight performance with synchronous safety.

6.1 Synchronous safety

We first evaluate how Aftersight performs when providing safety that is equivalent to running the analysis inline with the main workload. In this usage model, Aftersight runs the analysis in parallel with the main workload, and defers the output of the workload until the analysis reaches that output. Aftersight's main benefits for this usage model are the ability to add new analyses without fear of breaking the workload and the ability to conduct later analyses that were not envisioned at the time of the run.

When providing synchronous safety, Aftersight's performance is limited by the analysis VM. A reasonable expectation is that the performance of the analysis VM will be comparable to that of an inline system (or slower, due to replaying overhead). While this can be true for many workloads, the analysis VM in Aftersight can also run faster than an inline system by taking advantage of the work done by the primary VM (Section 4.2). We demonstrate an example of this phenomenon through the following experiment.

We evaluate Aftersight with synchronous safety on a workload that uses `wget` to fetch a directory of linked web pages from a local `lighttpd` web server. The directory of web pages consists of 5000 HTML files, each 200 KB. The workload starts with a cold file cache and spends most of its time fetching data from disk. The check running in the analysis VM mimics a trivial on-access virus scanner by computing for 2 ms on each disk request.

Figure 2 compares the performance for Aftersight with synchronous safety with running the analysis inline with the workload. The analysis VM in Aftersight always trails the primary VM that it is replaying, so the workload is considered complete when the analysis VM completes.

Although the analysis VM is slowed by the overhead of replaying, it leverages the disk reads performed by the primary VM to regain this performance. The net effect on this benchmark is that Aftersight achieves slightly better performance than inline analysis.

6.2 Best-effort safety

We next demonstrate how Aftersight enables heavy-weight analyses to execute concurrently with a workload with best-effort safety. Our analysis enforces protection for guest address spaces at the granularity of individual bytes of memory. This supports checking for a wide range of memory errors, though we only apply it to catching heap overflows in our example.

An in-memory bitmap specifies whether each byte of a particular address space is writable or not. The bitmap is organized as a two-level page table to conserve space. To implement the checks, the analysis dynamically instruments instructions that write to memory. These writes are translated to look up the appropriate protection bits in the bitmap and check if they allow writing. If they do, the write proceeds normally, otherwise the analysis invokes an error handler. When running the analysis in parallel with the main workload (online), the error handler can invoke a feedback action that takes corrective measures. For example, the system can automatically suspend the primary VM when an error is detected.

We implemented the analysis in VMware Workstation by modifying the binary translation done during replay. On each write, the translation saves two scratch registers and the CPU flags, checks the protection bits in a bitmap, then restores the two scratch registers.

To use this analysis, the guest kernel specifies the desired protection for each byte of memory of an address space. We modified a guest Linux kernel to use this facility to do heap-overflow bounds checks on dynamically allocated kernel objects. Linux already includes a facility for adding "red zone" buffers to the start and end of every `kmalloc` object. We use this facility and add code to set the bitmap permission bits for these redzones appropriately. Only kernel code needs checking, since normal page protections prevent user code from touching kernel heap objects.

Without Aftersight, this analysis is too slow to be used in production settings. We measured the speed of analysis without Aftersight by running the checks inline in the VMware binary translator of the main workload. The results are shown in Figure 3. For a kernel compilation benchmark, running the benchmark with the analysis inline (i.e. without Aftersight) takes 191.10 seconds (a 2.48x slowdown compared to running the benchmark without the analysis). Running the workload with Aftersight reduces the time to complete the benchmark to 84.86 seconds. With best-effort safety, Aftersight reduces the perturbation on the main workload by moving the overhead of analysis off the critical path of the main workload and into a separate analysis system. The main workload pays for this in the form of record/replay overhead, which for this benchmark is 10.4%.

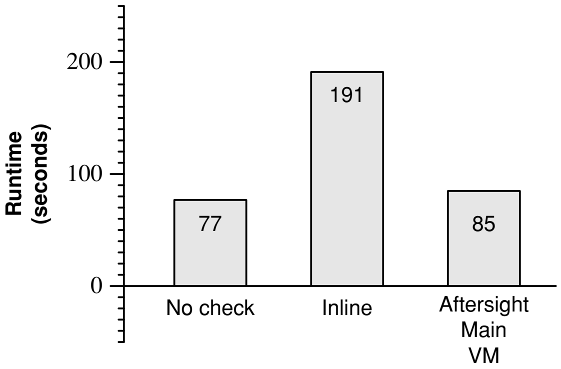

Figure 3: Comparison of kernel compile overhead with heap buffer overflow detection: Aftersight runs a kernel compilation benchmark nearly 2x faster than inline checking. Checks are still run concurrently with variable lag.

A potential disadvantage of decoupling this analysis from the main workload is that the detection of a write violation may occur long after the offending instruction execution. However, even delayed feedback can be very useful. For example, we ran an SSH server in the primary VM, logged into it from an outside client, and invoked a system call that contains an erroneous heap overflow. As before, we check byte-level write protections in an analysis VM. We measured how long it takes the analysis VM to discover the problem, assuming that prior to the SSH connection, both the primary and analysis VMs are synchronized to the same point in time.

In Figure 4 we see the progress of the primary and analysis VM, measured in #branches executed vs. wall clock time. As shown by the horizontal distance on the graph between identical branch counts in the primary and analysis VM, the analysis VM lags the primary VM by varying amounts during the run. In this experiment, there was a 0.86 second latency between the write violation in the primary VM and its detection in the analysis VM.

Decoupled analysis can lead to delays between an event in the main VM and the analysis of those events in the analysis VM. This delay is not a problem for analyses that don't need to provide feedback to the main VM, or for analyses that have no response time requirement on their feedback (such as optimization or bug finding). The delay is not ideal for analyses that implement security or correctness checks. However, delayed feedback can still be very useful, and is certainly preferable to being unable

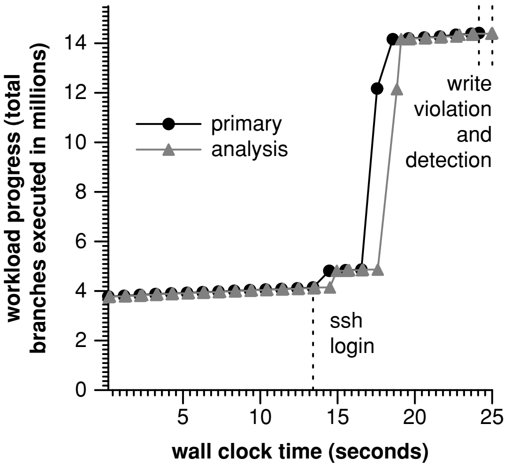

Figure 4: Measuring latency of simultaneous record and analysis for write protection violations: The two dotted lines at the end show when the write violation occurs and when it is identified by the analysis (shown separated by 0.86 seconds).

to run heavyweight analyses at all.

6.3 Idle-time boost

Section 6.2 showed that decoupling analysis from the main workload with best-effort safety can improve the performance of the main workload, but that the analysis may complete significantly after the workload. Since heavyweight analyses may be much slower than the main workload, a natural question is "How can heavyweight analyses keep up with the main workload over a long run?". Section 4.2 describes some techniques that help a replaying analysis VM to keep up with a main VM, and this section evaluates the effectiveness of these techniques. In particular, we observe that idle time in the primary can provide enough time for the analysis VM to catch up during replay, even for very heavyweight analyses.

As an example of a heavyweight analysis platform, we configure the VMware Workstation binary translator to translate *all* guest instructions (instead of just guest kernel instructions), but still with a relatively small (2.9 MB) code cache. Even with no extra analysis, this configuration of VMware Workstation runs several times slower than the normal configuration.

We measured the overhead of this analysis system on a CPU-bound workload, which is winLAME (a GUI frontend to the LAME mp3-encoder) in a Windows XP VM, encoding .wav files into mp3 format. The test machine is a two-processor, dual core 2GHz AMD Opteron (containing four logical CPUs) running Debian Linux. On this workload, the analysis VM takes 4.65x as long to complete the workload as the main VM, due to the overhead of the slower binary translator,

Most realistic workloads can be replayed much faster by skipping over idle time in the main workload. To see how idle time can provide a boost, we ran this analysis again with an interactive, desktop workload. In the primary VM, we use Windows XP and:

- Start Firefox. Edit the proxy settings to get out of the corporate network.
- Visit slashdot.org, scroll through the front page, browsing for one minute.
- Visit internal website and download an Excel spreadsheet containing numbers used in this paper.
- Close Firefox. Start Excel, open the spreadsheet.
- Create a chart, and plot two curves using data in the spreadsheet. Add a trend line to one of the curves.
- Close Excel. Open Powerpoint, and create a custom animation using four block arrows flying in from different directions. Close Powerpoint.

Figure 5 shows the results of this experiment. Figure 5(a) shows the progress of the primary VM and the analysis VM as wall clock time progresses. Horizontal gaps between the two curves occur where bursts of high CPU utilization cause analysis to lag the primary.

Figure 5(b) more clearly illustrates these bursts by showing the instantaneous compute rates of the primary and the analysis. Figure 5(b) shows that the primary contains many compute spikes. Meanwhile, the analysis runs at a more constant pace because it is limited by the speed of binary translation. These compute spikes can cause significant lag (one spike causes the main VM to execute 6x faster than the analysis VM). However, as shown in Figure 5(a), the idle times in the workload allows the analysis VM to eventual catch up from this lag.

Figure 5(c) graphs the lag between the main VM and the analysis VM as the workload progresses. This graph demonstrates two major benefits of decoupled analysis with best-effort safety. First, note that the analysis VM can lag behind the main VM significantly (10-11 seconds on average, and as high as 35 seconds behind). These periods of high lag imply that running the analysis inline with the main VM or with synchronous safety would cripple the interactive performance of the main VM (imagine waiting 35 seconds between clicking a button and waiting for the corresponding menu to appear!). In contrast, Aftersight with best-effort safety decouples the analysis, so that users of the primary VM don't experience this lag. Instead, the primary's responsiveness is completely independent of the speed of analysis, e.g. button clicks and menu selections occur at full speed.

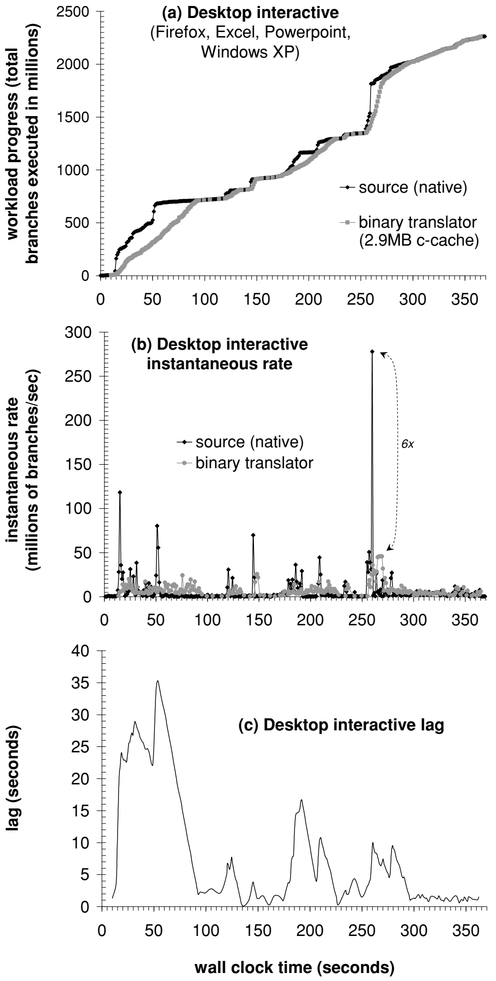

Figure 5: Plotting the progress of a source workload, and a online tandem analysis of that workload running on another core: a desktop interactive session shows how analysis (running in a binary translator that is several times slower) is allowed to lag during bursty periods of computing, and catch up when bursts end.

Second, this graph illustrates how idle times between bursts of CPU activity enable the analysis VM to catch up with the main VM. Although the lag gets as long as 35 seconds during this workload, the analysis VM is able to catch up by the end of the workload. Catching up in this manner is only possible because of decoupled analysis. Synchronizing the main VM with the analysis VM would limit the speed of the main VM during bursts of CPU activity to that of the analysis VM, and it would limit the speed of the analysis VM during idle periods to that of the (idling) main VM.

Idle time in real-world workloads is quite common. In an informal poll, idle time was over 95% for several desktop computers used by full-time computer programmers for compiling programs, editing tex, e-mail, web browsing, and running VMs. Idle time was over 75% for a production web server and mail server at the EECS department of a large public university.

Idle time can also be deliberately increased in many systems, and this may help heavyweight analyses keep up with the main VM. For example, idle time can be increased in server farms by adding more servers and balancing load across them.

6.4 Offline analysis

This section demonstrates how Aftersight enables heavyweight analyses to be built and applied after the main workload completes.

To illustrate this capability, we implemented an analysis that ensures every instruction executed by the source workload meets a set of memory safety guarantees: that a dereferenced pointer must point to valid stack or heap data, that any bit of stack or heap data used in control flow or as a pointer/index must be initialized before use, and that there are no memory leaks (roughly, that the guest executes "Valgrind-safe" [26]).

Asserting continual satisfaction of these constraints is a very expensive job—properly implementing a check for uninitialized data requires a full taint propagation analysis [26]. Our tool implements this taint analysis and does it at bit-level precision, largely following the implementation given by [26]. The analysis tracks the state of each bit in all guest memory and registers according to the state machine shown in Figure 6. To initialize the state of each bit, the checker interposes on all memory allocation requests for the heap (through calls to memory allocator functions) and stack (through manipulations of the stack pointer). To maintain the value of the state of each bit, the checker interposes on every instruction executed by the workload, for example to propagate the state of source memory or registers to destination memory or registers. The tool uses symbol information to identify calls to the appropriate heap allocator for the system. When analyzing a particular user process, the target process first identifies itself by making a hypercall.

The analysis is built using Aftersight's QEMU-based CPU simulator. The analysis is extremely heavyweight (on the order of 100x) and is best suited to running offline.

Implementing this analysis in Aftersight yields two important benefits relative to traditional tools such as Valgrind and Purify. First, Valgrind and Purify slow their target program too much to be used for long-running, realistic workloads, and they may perturb their target programs too much to capture realistic interactive workloads. In contrast, Aftersight allows long-running, interactive workloads to run with little overhead by allowing the work of analysis to be run later. Second, tools such

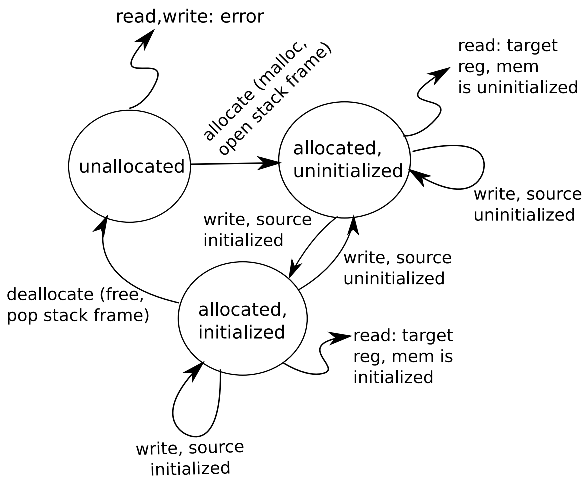
read,write: error
allocate (malloc, open stack frame)
read: target reg, mem is uninitialized
unallocated
allocated, uninitialized
write, source uninitialized
write, source initialized
write, source uninitialized
deallocate (free, pop stack frame)
allocated, initialized
read: target reg, mem is initialized
write, source initialized

Figure 6: Memory consistency checker: a state machine is associated with every bit of guest memory and registers, and finds uses of garbage data and dangling pointers.

as Valgrind and Purify can only be applied to user-level code. They cannot be applied to OS or VMM code, even though such code is critical to reliability and safety. In contrast, analyses implemented in Aftersight can be applied to all software running in the virtual machine, including the operating system or even another VMM running inside VMware Workstation.

We have used this tool to find serious bugs in large, complex systems, including kernel code such as VMware ESX Server and Linux. The rest of this section describes bugs we found with this analysis.

ESX Server We used our analysis tool in the development of VMware ESX Server [30] by running ESX inside a VM hosted by VMware Workstation. We found 10 type safety errors, over half of which were classified as critical or show-stopper bugs, and were able to fix them during development.

For example, in one bug the ESX Server kernel has a utility data structure for recording statistics whose use is sprinkled throughout the code. It takes an array of values as argument, and stores it:

```
Histogram_New(..., const uint32 numBuckets,
              const Histogram_Datatype*
                  const bucketLimits) {
   ...
   histo = Heap_Alloc(heap,
               sizeof(struct Histogram) +
                         bucketCountsSz);
   if (histo != NULL) {
      histo->numBuckets = numBuckets;
      ...
      histo->limits.arbitrary.bucketLimits =
         bucketLimits;
   }
```

```
   ...
```

This array of values is used when manipulations to the statistics occur. Unfortunately, the structure is allocated on the heap, and some callers initialize it with an array from the stack:

```
SCSIAllocStats(Heap_ID heap,
               ScsiStats *stats) {
   Histogram_Datatype  limits[...];
   ...
   stats->cmdSizeHisto =
      Histogram_New(..., limits);
   ...
```

Under certain circumstances, this would cause a crash, but the tool was able to diagnose the problem without reproducing the crash by noticing that using the data structure caused references to data located in popped off stack frames.

Linux We also applied our analysis tool to the Linux kernel by running it as the guest kernel in a VM. Our tool diagnosed a long-overlooked type safety error in an old part of the core Linux kernel. Its UDP stack makes use of uninitialized stack garbage on reception of UDP packets through `recvfrom`. Whenever `recvfrom` is called, a `msg` structure containing a field `msg_flags` would be allocated on the stack, but never initialized:

```
/* from net/socket.c */
asmlinkage long sys_recvfrom(...) {
  ...
  struct msghdr msg;
  ...
  msg.msg_control = NULL;
  msg.msg_controllen = 0;
  msg.msg_iovlen = 1;
  msg.msg_iov = &iov;
  iov.iov_len = size;
  iov.iov_base = ubuf;
  msg.msg_name = address;
  msg.msg_namelen = MAX_SOCK_ADDR;
  if (sock->file->f_flags & O_NONBLOCK)
    flags |= MSG_DONTWAIT;
  err = sock_recvmsg(sock, &msg, size, flags);
  ...
```

Following the call stack down through `sock_recvmsg`, this structure is passed to `udp_recvmsg`, which uses `msg_flags`:

```
Backtrace:
#0 udp_recvmsg
        (linux-2.6.20.1/net/ipv4/udp.c:843)
#1 sock_common_recvmsg
      (linux-2.6.20.1/net/core/sock.c:1617)
#2 sock_recvmsg
        (linux-2.6.20.1/net/socket.c:630)
#3 sys_recvfrom
        (linux-2.6.20.1/net/socket.c:1608)
#4 sys_socketcall
        (linux-2.6.20.1/net/socket.c:2007)
```

```
#5 syscall_call
(linux-2.6.20.1/arch/i386/kernel/entry.S:0)

/* net/ipv4/udp.c */
int udp_recvmsg(..., struct msghdr *msg, ...)
{
  ...
  if (.. &&  msg->msg_flags&MSG_TRUNC)) {
  ...
```

In this case, the test on `msg_flags` gated a checksum computation, so although no crash would result from this erroneous use of `msg_flags`, it would cause random, unnecessary extra computation to occur on reception of UDP packets.

We discovered this bug in Linux 2.6.20.1 and reported it to kernel developers, who fixed it in the next major release. This bug existed in the code for many years (all prior versions of 2.6 and all versions of 2.4 we checked back to 2002).

Putty We also tested a common Windows SSH client called Putty. Our tool found one memory leak that was invoked whenever a menu item was selected. We are currently investigating other user programs as well.

7 Future work

There are many interesting open questions about how to optimize and synchronize record and analysis.

Workload memoization Memoizing state generated by native hardware during record can avoid the need for re-computation during analysis, and this can be used to accelerate analysis. We use this kind of memoization when simulating SMM (system management) mode [13]. Our CPU simulator does not implement *x*86's SMM mode because the version of QEMU we started with did not support it. However, the VMware VMM is more faithful about this part of the architecture, and some target workloads do contain execution in SMM. Aftersight memoizes SMM to maintain compatibility for these workloads. In its relogging step, Aftersight records the changes to memory made in SMM (some of which may be used outside SMM, for example by the guest BIOS code). During replay, the analysis infrastructure reproduces these effects at the proper time, thus avoiding the need to simulate SMM code.

In addition to helping compatibility, memoization can also be used to accelerate replay. For example, consider an analysis where we are only interested in the execution of a specific user process. With memoization, we could use relogging to summarize the execution of all other code in the system into their effects on memory and registers. In essence, this turns the execution of the OS and other processes into the equivalent of a single DMA operation. This would allow subsequent replay and analyses to ignore writes to pages not mapped into the current process, context switches to other processes, and most kernel activity. Focusing on one process would also allow us to accelerate the simulator by not emulating the hardware MMU. Instead we could simply use `mmap` or its equivalent to set up the address space for the process and allow memory accesses to run natively.

Feedback modes Simultaneous record and analysis mode can use different types of feedback loops to synchronize. We discussed the tradeoffs between two basic approaches, blocking and lazy feedback modes, in Section 4. However, one limitation of these approaches is that they fail to take into account the semantics of the OS, application, or analysis.

If we add more intelligence to our synchronization strategy we can take advantage of natural join points that occur. For example, when analyzing a web server for security, we can impose a synchronization restriction that the analysis VM must be in-synch with the primary whenever the primary initiates an *outgoing* TCP connection (delaying the primary, if necessary, to guarantee the synchronization), assuming we expect such events to be important but relatively rare. Synchronizing on such an event provides a hard guarantee that can prevent the spread of an attack, yet maximizes the amount of time the analysis machine has to catch up with the primary.

8 Related work

Replay facilities in a VMM have been discussed by a number of researchers [4, 9, 35] and used for a variety of purposes. For example, Bressoud and Schneider log non-determinism to support re-execution of a whole machine (OSes and applications) and use this to tolerate fail-stop faults on HP PA-RISC [4]. ReVirt [9] uses VM replay on *x*86 systems to enable *ex post facto* analysis for computer forensics. Aftersight uses VM replay for another purpose, which is to enable heavyweight dynamic analysis to be used on realistic workloads without perturbing them. Aftersight is also more flexible than these past VM replay systems because it allows analyses to run in a different environment from the primary, such as a simulator. Aftersight also leverages the fact that replicas can run faster than the primary to make online analysis more practical.

Researchers have suggested using replay implemented at the virtual-machine level or in hardware to conduct various types of offline analyses, such as computer forensics [9, 14], debugging [15, 16, 34], and architectural simulation [35]. Aftersight makes more types of analysis practical by allowing the analysis to run in a simulator, which reduces the cost of context switching between the replaying virtual machine and the analysis code. Analyses like taint analysis, which require frequent switches, are impractical without this capability. Aftersight also extends the use of VM replay to both online and offline analysis.

Other researchers have sought to run analyses online and in parallel with the original program via software or hardware support. Patil and Fischer proposed running an instrumented "shadow process" in parallel with the original program [21] and used this approach to implement memory safety checks. Speck [19] and SuperPin [31] fork multiple analysis processes from an uninstrumented process, using record/replay to synchronize the analysis processes. Whereas these past approaches can only analyze an application process, Aftersight expands the scope of decoupled analysis to include the operating system and all applications running on a machine. Aftersight also uses a more complete replay system that handles asynchronous interrupts.

Oplinger and Lam leverage proposed hardware support for thread-level speculation (TLS) to enable the original program to run in parallel with monitoring code [20]. They depend on proposed hardware support for TLS to detect data dependencies and rollback the original program if it causes a conflict. Similarly, Zhou, et al. use proposed hardware support for TLS to run memory-monitoring functions in parallel with the original program [36]. In contrast to this prior work, Aftersight requires no hardware support and works on today's commodity processors. Without support for TLS, current processors cannot quickly fork a new thread. Instead, Aftersight uses virtual-machine replay to continuously mirror the dynamic state of the original program on the analysis machines, thereby making it possible for them to analyze this state on spare processors or cores. Aftersight also includes new optimizations to accelerate the analysis machines by leveraging information generated by the original program.

A recent workshop paper briefly describes a similar approach to executing analyses in parallel with the original program [6]. As with TLS, their system requires hardware support to mirror the dynamic state of the original program onto spare processors by logging detailed data from each instruction, including the instruction counter, type, and input/output identifiers. The large volume of log data slows performance down by 4-10x. In contrast, Aftersight requires no hardware support, and runs with low overhead.

Other research has looked at combining simulators and hypervisors. For example, Ho, et al. [12] allowed switching back and forth between QEMU and Xen, and SimOS allows users to switch between direct execution and detailed simulation [22]. Aftersight differs from these systems by decoupling the analyses from the main workload and allowing both modes to run at the same time. This takes advantage of parallelism in processor cores and eliminates the user-visible overheads of analysis.

Besides decoupled analysis, there are a variety of other ways to reduce the overhead of heavyweight analysis. For example, sampling reduces analysis overhead [7] but can miss relevant events and only works for specific analyses. Hardware solutions [29] can also reduce the perturbation to the main workload causes by analysis, but requiring custom hardware makes this approach less attractive.

9 Conclusions

Dynamic program analysis has a wide range of compelling uses. Unfortunately, powerful analyses typically add substantial overhead which perturbs the workload, so the vast majority of program execution takes place with very little checking. This means that many critical software flaws remain overlooked, when they could be detected during testing, quality assurance, and deployment. Similarly, in operational settings, the high overhead of analysis deters the use of many potentially promising techniques for intrusion detection and prevention.

We have presented *Aftersight*, a system that helps overcome these limitations by decoupling dynamic program analysis from execution through virtual machine replay. This allows analysis to be carried out on the replayed execution, independent of the main workload. This mechanism allows several choices along the safety/performance spectrum, such as synchronous safety, best-effort safety, and offline analysis. Synchronous safety achieves performance comparable to inline analysis, while best-effort safety and offline analysis make it possible to apply slow, expensive analysis techniques on realistic production workloads without perturbing their performance.

We discussed how Aftersight supports the use of different record and replay platforms and the benefits of allowing each to be independently optimized based on their need for performance or extensibility. We presented our prototype of Aftersight, and evaluated it with several online and offline analyses.

Dynamic program analysis is a promising technique for solving many problems. However, without a means of overcoming its performance costs, it will continue to see limited use. In light of the ubiquitous adoption of virtualization technology, we believe decoupled analysis, as demonstrated by Aftersight, offers a promising approach to enabling the use of this technique in a much broader set of applications.

References

[1] K. Adams and O. Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *ASPLOS*, pages 2–13, October 2006.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[3] F. Bellard. QEMU, A Fast and Portable Dynamic Translator. In *Proc. USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[4] T. C. Bressoud and F. B. Schneider. Hypervisor-based Fault Tolerance. *ACM Trans. Comput. Syst.*, 14(1):80–107, 1996.

[5] F. Chang and G. A. Gibson. Automatic I/O hint generation through speculative execution. In *Proceedings of the 1999 Symposium on Operating Systems Design and Implementation (OSDI)*, February 1999.

[6] S. Chen, B. Falsafi, P. B. Gibbons, M. Kozuch, T. C. Mowry, R. Teodorescu, A. Ailamaki, L. Fix, G. R. Ganger, B. Lin, and S. W. Schlosser. Log-Based Architectures for General-Purpose Monitoring of Deployed Code. *2006 Workshop on Architectural and System Support for Improving Software Dependability*, October 2006.

[7] T. M. Chilimbi and M. Hauswirth. Low-Overhead Memory Leak Detection Using Adaptive Statistical Profiling. In *ASPLOS*, October 2004.

[8] J.-D. Choi and H. Srinivasan. Deterministic replay of Java multithreaded applications. In *Proc. 1998 SIGMETRICS Symposium on Parallel and distributed tools (SPDT)*, August 1998.

[9] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *OSDI*, pages 211–224, New York, NY, USA, 2002. ACM.

[10] S. L. Graham, P. B. Kessler, and M. E. McKusick. Gprof: A Call Graph Execution Profiler. In *SIGPLAN '82 Symposium on Compiler Construction*, pages 120–126, June 1982.

[11] R. Hastings and B. Joyce. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In *Proc. Winter 1992 USENIX Conference*, pages 125–138, 1992.

[12] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical Taint-Based Protection Using Demand Emulation. In *EuroSys*, pages 29–41, New York, NY, USA, 2006. ACM Press.

[13] Intel. IA-32 Intel Architecture Software Developer's Manual. Volumes I, II, and III, 2006.

[14] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *SOSP*, pages 91–104, October 2005.

[15] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging Operating Systems with Time-Traveling Virtual Machines. In *Proc. USENIX Annual Technical Conference*, pages 1–15, 2005.

[16] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *ISCA*, pages 284–295, Washington, DC, USA, 2005. IEEE Computer Society.

[17] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *PLDI*, pages 89–100, New York, NY, USA, 2007. ACM Press.

[18] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proc. Network and Distributed System Security Symposium (NDSS)*, 2005.

[19] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing Security Checks on Commodity Hardware. In *ASPLOS*, March 2008.

[20] J. Oplinger and M. S. Lam. Enhancing Software Reliability using Speculative Threads. In *ASPLOS*, pages 184–196, October 2002.

[21] H. Patil and C. N. Fischer. Efficient Run-time Monitoring Using Shadow Processing. In *Proc. International Workshop on Automated and Algorithmic Debugging (AADEBUG)*, pages 119–132, May 1995.

[22] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the SimOS Machine Simulator to Study Complex Computer Systems. *Modeling and Computer Simulation*, 7(1):78–103, 1997.

[23] M. Rosenblum and T. Garfinkel. Virtual Machine Monitors: Current Technology and Future Trends. *IEEE Computer*, May 2005.

[24] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[25] B. A. Schroeder. On-Line Monitoring: a Tutorial. *IEEE Computer*, 28(6):72–78, June 1995.

[26] J. Seward and N. Nethercote. Using Valgrind to Detect Undefined Value Errors With Bit-Precision. In *Proc. USENIX Annual Technical Conference*, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.

[27] S. Srinivasan, S. Kandula, C. Andrews, and Y. Zhou. Flashback: A light-weight rollback and deterministic replay extension for software debugging. In *Proc. USENIX Technical Conference*, June 2004.

[28] 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*, pages 1–14, 2001.

[29] J. J. P. Tsai, K.-Y. Fang, H.-Y. Chen, and Y.-D. Bi. A Noninterference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging. *IEEE Transactions on Software Engineering*, 16(8):897–916, August 1990.

[30] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *OSDI*, pages 181–194, December 2002.

[31] S. Wallace and K. Hazelwood. SuperPin: Parallelizing Dynamic Instrumentation for Real-Time Performance. In *Proc. 2007 International Symposium on Code Generation and Optimization (CGO)*, pages 209–217, March 2007.

[32] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. In *OSDI*, December 2004.

[33] E. Witchel and M. Rosenblum. Embra: Fast and Flexible Machine Simulation. *SIGMETRICS Perform. Eval. Rev.*, 24(1):68–79, 1996.

[34] M. Xu, R. Bodik, and M. D. Hill. A flight data recorder for enabling full-system multiprocessor deterministic replay. In *ISCA*, pages 122–135, New York, NY, USA, 2003. ACM Press.

[35] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. ReTrace: Collecting Execution Trace with Virtual Machine Deterministic Replay. In *Proc. 2007 Workshop on Modeling, Benchmarking and Simulation (MoBS)*, June 2007.

[36] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient Architectural Support for Software Debugging. In *ISCA*, June 2004.