

Profiling and Controlling Nondeterminism in Linux Services

by

Syed Aunn Hasan Raza

S.B., Computer Science and Engineering, M.I.T., May 2010

Submitted to the Department of Electrical Engineering and Computer
Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2011

© Massachusetts Institute of Technology 2011. All rights reserved.

Author
Department of Electrical Engineering and Computer Science
August 15, 2011

Certified by
Dr. Saman P. Amarasinghe
Professor
Thesis Supervisor

Accepted by
Dr. Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Profiling and Controlling Nondeterminism in Linux Services

by

Syed Aunn Hasan Raza

Submitted to the Department of Electrical Engineering and Computer Science
on August 15, 2011, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Both server and desktop virtualization rely on high VM density per physical host to reduce costs and improve consolidation. In the case of *boot-storms*, such high VM density per host can be a problem....

(To be filled in)

Thesis Supervisor: Dr. Saman P. Amarasinghe

Title: Professor

Acknowledgments

I would like to thank Professor Saman Amarasinghe for his huge role in both this project and my wonderful undergraduate experience at MIT. Saman was my professor for 6.005 (Spring 2008), 6.197/6.172 (Fall 2009) and 6.035 (Spring 2009). These three exciting semesters not only convinced me of his unparalleled genius, but also ignited my interest in computer systems. Over the past year, as I have experienced the highs and lows of research, I have really benefited from Saman's infinite insight, encouragement and patience.

As an M-Eng student, I have been blessed to work with two truly inspirational and gifted people from the COMMIT group: Marek Olszewski and Qin Zhao. Their expertise and brilliance is probably only eclipsed by their humility and helpfulness. I have learned more from Marek and Qin than I probably realize, and their knowledge of operating systems and dynamic instrumentation is invaluable and, frankly, immensely intimidating. I hope to emulate (or even approximate) their excellence some day.

This past year, I have also had the opportunity to work with Professor Srinivas Devadas, Professor Fraans Kaashoek, and Professor Dina Katabi as a TA for 6.005 and 6.033. It has been an extraordinarily rewarding experience, and I have learned tremendously from simply interacting with these peerless individuals. Professor Dina Katabi was especially kind to me for letting me work in G916 over the past few months.

I would like to thank Abdul Munir, whom I have known since my first day at MIT; I simply don't deserve the unflinchingly loyal and supportive friend I have found in him. I am also indebted to Osama Badar, Usman Masood, Brian Joseph, and Nabeel Ahmed for their unrelenting support and encouragement; this past year would have been especially boring without our never-ending arguments and unproductive 'all-nighters'. I also owe a debt of gratitude to my partners-in-crime Prannay Budhraj, Ankit Gordhandas, Daniel Firestone and Maciej Pacula, who have been great friends and collaborators over the past few years.

I am humbled by the countless sacrifices made by my family in order for me to be where I am today. My father has been the single biggest inspiration and support in my life since childhood. He epitomizes, for me, the meaning of selflessness and resilience. This thesis, my work and few achievements were enabled by – and dedicated to – him, my mother and my two siblings Ali and Zahra. Ali has been a calming influence during my years at MIT; the strangest (and most unproductive) obsessions unite us, ranging from Chinese *Wuxia* fiction to, more recently, *The Game of Thrones*. Zahra’s high-school math problems have been a welcome distraction over the past year; they have also allowed me to appear smarter than I truly am.

Finally, I would like to thank my wife Amina for her unwavering love and support throughout my stay at MIT, for improving and enriching my life every single day since I have known her, and for knowing me better than even I know myself. Through her, I have also met two exceptional individuals, Drs. Fatima and Anwer Basha, whom I have already learnt a lot from.

“It is impossible to live without failing at something, unless you live so cautiously that you might as well not have lived at all – in which case, you fail by default.”

J.K. Rowling, Harvard Commencement Speech 2008

Contents

1	Introduction	13
1.1	Motivation	13
1.2	Goal of Thesis	15
1.3	Contributions	15
1.4	Importance of Deterministic Execution	16
1.5	Thesis Organization	17
2	Execution Profile of Linux Services	19
2.1	The Linux Boot Process	19
2.2	Data Collection Scheme	22
2.2.1	Measuring nondeterminism in a simple <i>C</i> program	23
2.2.2	Quantifying Nondeterminism	28
2.3	Results for Linux services	29
2.4	Summary	33
3	Overcoming Nondeterminism in Linux services	35
3.1	Sources of Nondeterminism	35
3.1.1	Linux Security Features	36
3.1.2	Randomization Schemes	37
3.1.3	Process Identification Layer	39
3.1.4	Time	40
3.1.5	File I/O	41
3.1.6	Network I/O	42

3.1.7	Scalable I/O Schemes	43
3.1.8	Signals	45
3.1.9	Concurrency	45
3.1.10	<i>Procfs</i> : The ‘/proc/’ directory	46
3.1.11	Architecture Specific Instructions	46
3.2	Simulating Deterministic Execution	47
3.3	Limitations of Deterministic Execution	47
3.4	Summary	48
4	<i>Silhouette</i> Execution	49
4.1	What is <i>Silhouette</i> execution?	49
4.2	Simulation Scheme	51
4.3	Evaluation of Simulation	59
4.4	Summary	59
5	Design Ideas for Deduplication of Execution	61
5.1	Summary	61
6	Conclusion and Future Work	63
6.1	Summary	63

List of Figures

2-1	CPU and disk activity for a booting Ubuntu VM in the first 35 seconds after <code>init</code> is spawned	20
2-2	(... continued) CPU and disk activity for a booting VM 35 seconds after <code>init</code> is spawned.	20
2-3	A summary of the actions performed by <code>init</code> for a booting VM . . .	21
2-4	Steps involved in measuring execution nondeterminism	22
2-5	A “Hello, world!” program in C.	23
2-6	Excerpts from the log files generated by the execution tracing layer .	24
2-7	Excerpts from the side-by-side diff files generated by the analysis script	25
2-8	Visualization of “Hello, world!” program execution	27
2-9	The cascade and propagation effects in measuring nondeterminism. .	28
2-10	Visualization of <code>ntp</code> program execution	31
2-11	Visualization of <code>cron</code> program execution	31
2-12	Understanding nature of conflicts in <code>cron</code>	32
4-1	Simulation of Silhouette Execution in a bootstorm scenario	52
4-2	Virtualizing the process ID layer using Pin	55
4-3	Reordering I/O events using Pin	57

List of Tables

2.1	Nondeterminism profile of “Hello, world!” program (ASLR disabled) .	27
2.2	Nondeterminism profile of Linux services and daemons (ASLR disabled)	30
2.3	Measuring burstiness of nondeterminism in Linux services	32

Chapter 1

Introduction

1.1 Motivation

Data centers increasingly use server virtualization to reduce operating costs, simplify administrative tasks and improve performance scalability. Through virtualization, it is possible to achieve high resource utilization and isolation at the same time: each application is typically assigned a dedicated server virtual machine (VM), while many VMs are consolidated on powerful host computers to reduce wasted cycles. The use of techniques such as memory overcommitment (including *transparent page sharing*, *ballooning* and *hypervisor swapping*) [18] has further improved the consolidation ratios and cost-effectiveness of server virtualization, and augurs well for the future of the technology.

Given the success of server virtualization, many companies are extending the use of virtualization to their desktop computers. In a Virtual Desktop Infrastructure [17] (VDI), desktop operating systems and applications are hosted in virtual machines that reside in a data center; users access virtual desktops from desktop PCs or thin clients via a remote display protocol. A VDI provides simplicity in administration and management: applications can be centrally added, deleted, upgraded and patched. VDI deployments also promise even higher consolidation ratios than those achieved via server virtualization because desktop virtual machines typically require less resources than server virtual machines.

Consolidation ratios (measured by VM density per host) in data centers are expected to increase in the future, not only because of improvements in virtualization technology, but also because newer generations of processors support more cores and memory [4]. Because a single VM would typically utilize only a modest fraction of a host’s hardware resources, a high VM density per host is desirable in most cases for effective resource utilization. However, correlated spikes in the CPU/memory usage of many VMs can suddenly cripple host machines. For instance, a *boot storm* [4, 6, 9, 14, 16] can occur after some software is installed or updated, requiring hundreds or thousands of identical VMs to reboot at the same time. Bootstorms can be particularly frequent in VDIs because users typically show up to work at roughly the same time in the morning each day.

Concurrently booting VMs create unusually high I/O traffic, generate numerous disk and memory allocation requests, and can saturate host CPUs. To avoid the prohibitively high boot latencies that result from boot storms, data centers usually either boot machines in a staggered fashion, or invest in specialized, expensive and/or extra-provisioned hardware for network/storage [5, 6]. There is also anecdotal evidence that VDI users sometimes leave their desktop computers running overnight to avoid morning boot storms; this practice represents an unnecessary addition to already exorbitant data center energy bills [13]. Data deduplication [3], through which hosts reclaim/reuse disk blocks common to several VMs, has been proven to reduce the memory footprint of concurrently booting machines. However, while data deduplication can mitigate the stress on the memory subsystem in a boot storm, lowered memory latency can in turn overwhelm the CPU, fibre channel, bus infrastructure or controller resources and simply turn them into bottlenecks instead [10].

With the spread of virtualization, it is important to address the bootstorm problem in a way that does not involve simply skirting around the issue. Data deduplication is partly effective because identical VMs load the same data from disk when they boot up. In this thesis, we pose the following question: is it possible to generalize deduplication of data to deduplication of *execution*? If many identical VMs are concurrently booting up in a data center, do they execute the same set of instructions?

Even if there are some differences in the instructions executed, are they caused by controllable sources of non-determinism? Ultimately, if there is a way to ensure that concurrently booting VMs execute mostly the same set of instructions and perform the same I/O requests, one way to solve the boot storm problem may be remarkable simple in essence: instead of booting N identical VMs concurrently, we can boot one VM as a leader; the remaining $(N - 1)$ VMs follow the leader by executing a tiny subset of the instructions they would otherwise execute; we split execution into N different instances as late as possible into the boot process. This approach could potentially reduce pressure on the underlying host hardware, and thereby enable data centers to handle boot storms effectively.

1.2 Goal of Thesis

This thesis aims to address the following questions:

1. When identical VMs boot up concurrently, how similar are the sets of instructions executed? What is the statistical profile of any differences in the executed instructions?
2. What are the source(s) of any differences in the instruction streams of concurrently booting VMs? Are there ways to minimize the non-determinism in booting VMs?

The answers to these questions are clearly crucial in determining the feasibility of *deduplication of execution* as a possible solution to the boot storm problem.

1.3 Contributions

For this work, we used dynamic instrumentation frameworks such as DynamoRio [2] and Pin [7] to study user-level instruction streams from a few representative Linux services at boot-time.

In this document, we:

1. quantify nondeterminism in Linux services, and show that it is bursty and rare;
2. document the sources of nondeterminism in Linux services – both obvious and obscure – and specify strategies for overcoming them in the boot storm scenario;
3. use simple dynamic instrumentation techniques to show that *fully* deterministic execution is achievable without *any* modifications to Linux or an executing service.

Strategies to achieve deterministic execution have been studied at the operating system layer [1] before, but they require modifications to Linux. Deterministic execution can be achieved in multi-threaded programs using record-and-replay approaches [12] or deterministic logical clocks [11]. Our study of non-determinism has different goals from both approaches: we wish to avoid changing existing software (to ease adoption); we also wish to make several distinct – and potentially different – executions *overlap* as much as possible, rather than replay one execution over and over. In our case, we do not know *a priori* whether two executions will behave identically or not. That the behavior of system calls or signals in Linux can lead to different results or side-effects across multiple executions of an application is well known: what is not documented is the application *context* in which these sources of nondeterminism originate. To the best of our knowledge, this is the first attempt to study the statistical profile and context of nondeterminism in Linux services in such detail. While we hope this work ultimately proves the basis for an implementation of our proposed solution to the boot storm problem, we also note that deterministic execution can immediately improve the effectiveness of existing virtualization technologies such as transparent page sharing and data deduplication.

1.4 Importance of Deterministic Execution

While our study of nondeterminism is driven by a specific application, deterministic execution is desirable in a variety of scenarios. The motivations for deterministic

multithreading listed in [11, 12] apply to our work as well.

Mainstream Computing, Security and Performance: If distinct executions of the same program can be expected to execute the same set of instructions, then any significant deviations can be used to detect security attacks. Runtime detection of security attacks through the identification of anomalous executions is the focus of *mainstream computing* [15], and deterministic execution obviously helps in reducing false positives. Anomalous executions can also be flagged for performance debugging.

Testing: Deterministic execution in general facilitates testing, because outputs and internal state can be checked at certain points with respect to expected values. Our version of determinism allows for a particularly strong kind of test case that may be necessary for safety-critical systems: a program must execute the exact same instructions across different executions (for the same inputs).

Debugging: Erroneous behavior can be more easily reproduced via deterministic execution, which helps with debugging. Deterministic execution has much lower storage overhead than traditional record-and-replay approaches.

1.5 Thesis Organization

In what follows, Chapter 2 presents an overview of the Linux boot process, along with the dynamic instrumentation techniques we used to profile non-determinism in Linux services. Chapter 3 presents a summary of the sources of nondeterminism discovered in this work and the strategies we used to eliminate them. Chapter ?? presents a detailed case study of three Linux services to identify the common context in which non-determinism arises. Chapter 5 presents design ideas for an implementation of deduplication of execution. Finally, Chapter 6 concludes this thesis and discusses future work.

Chapter 2

Execution Profile of Linux Services

This chapter provides some background on the Linux startup process (Section 2.1). It then describes how we collected user-level instruction streams from some Linux services via dynamic instrumentation to measure nondeterminism in the Linux boot process (Section 2.2). Finally, it summarizes our results on the statistical nature of nondeterminism in Linux services (Section 2.3).

2.1 The Linux Boot Process

When a computer boots up:

1. The BIOS (Basic Input/Output System) gets control and performs startup tasks for the specific hardware platform.
2. Next, the BIOS reads and executes code from a designated boot device that contains part of a Linux boot loader. Typically, this smaller part (or phase 1) loads the bulk of the boot loader code (phase 2).
3. The boot loader may present the user with options for which operating system to load (if there are multiple available options). In any case, the boot loader loads and decompresses the operating system into memory; it sets up system hardware and memory paging; finally, it transfers control to the kernel's `start_kernel()` function.

4. The `start_kernel()` function performs the majority of system setup (including interrupts, remaining memory management, device initialization) before spawning the `idle` process, the scheduler and the user-space `init` process.
5. The scheduler effectively takes control of system management, and kernel stays idle from now on unless externally called.
6. The `init` process executes scripts that set up all non-operating system services and structures in order to allow a user environment to be created, and then presents the user with a login screen.

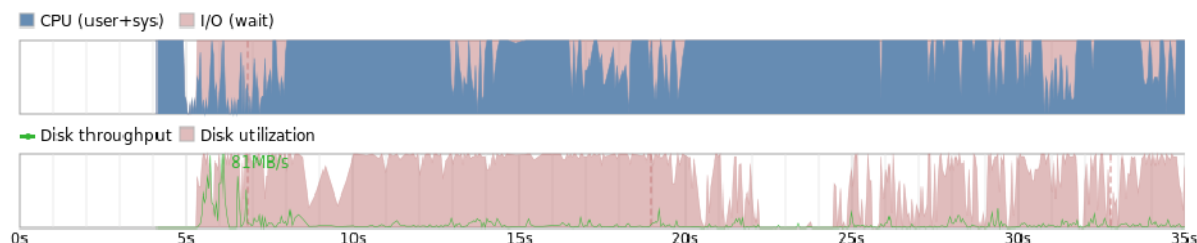


Figure 2-1: CPU and disk activity for a booting Ubuntu VM in the first 35 seconds after `init` is spawned. The first few seconds show no activity because the data collection daemon takes a few seconds to start.

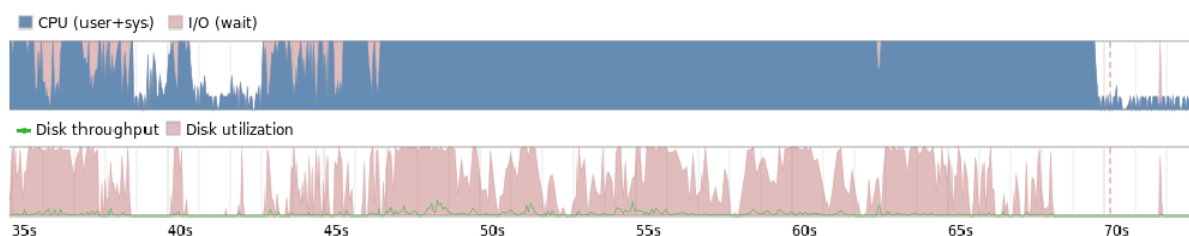


Figure 2-2: (... continued from Figure 2-1) CPU and disk activity for a booting Ubuntu VM 35 seconds after `init` is spawned.

Figures 2-1 and 2-2 illustrate the CPU usage and disk activity of an Ubuntu 10.10 VM that takes about 70 seconds to complete the sixth step of the boot process (i.e. spawning the `init` process to set up the user environment). The Linux kernel version is 2.6.35-27-generic and the VM is configured with a single core processor with 512 MB RAM. Generated using the Bootchart utility [8], the figures illustrate that the booting process involves high memory and CPU overhead (5-70 seconds); they also

show a glimpse of the well-known fact that memory and CPU overhead typically diminishes greatly after the boot process is completed and the machine is ready for login (70+ seconds). This disparity in CPU/memory usage is the source of the boot storm problem; a single host can handle many VMs in steady-state usage but gets crippled when the same VMs boot up concurrently.

In the last step of the booting process (step 6), `init` typically runs many scripts located in specific directories such as `/etc/rc` or `/etc/init.d/`. While the myriad Linux distributions can have their own variants of `init` binaries (e.g. `SysV`, or `systemd` or `Upstart`), the `init` process always directly/indirectly launches several services and daemons to initialize the user desktop environment. Figure 2-3 provides a summary of the specific actions performed by `init` (through the subprocesses or daemons it launches) for the same Ubuntu VM used for Figures 2-1 and 2-2.

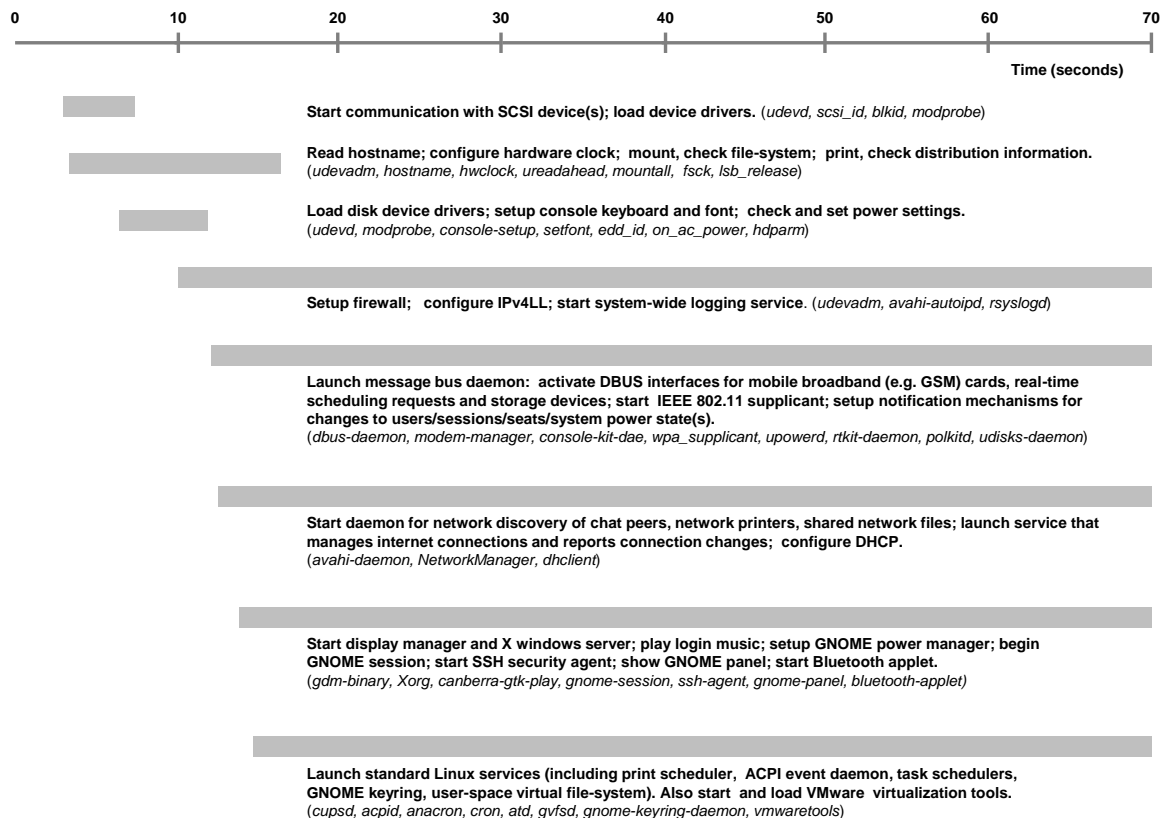


Figure 2-3: A summary of the actions performed by `init` for a booting VM; this figure has the same timeline (0-70 seconds) as Figures 2-1 and 2-2.

In fact, the `init` process actually launched 361 children processes (directly and indirectly) over the 70 second period summarized by Figure 2-3. Most of them were ephemeral processes; several processes were repeatedly launched in different contexts (e.g. `getty` or `grep`). The processes singled out in Figure 2-3 are the ones that either stayed alive through most of the boot process till the end, performed important boot actions, or spawned many sub-processes themselves.

2.2 Data Collection Scheme

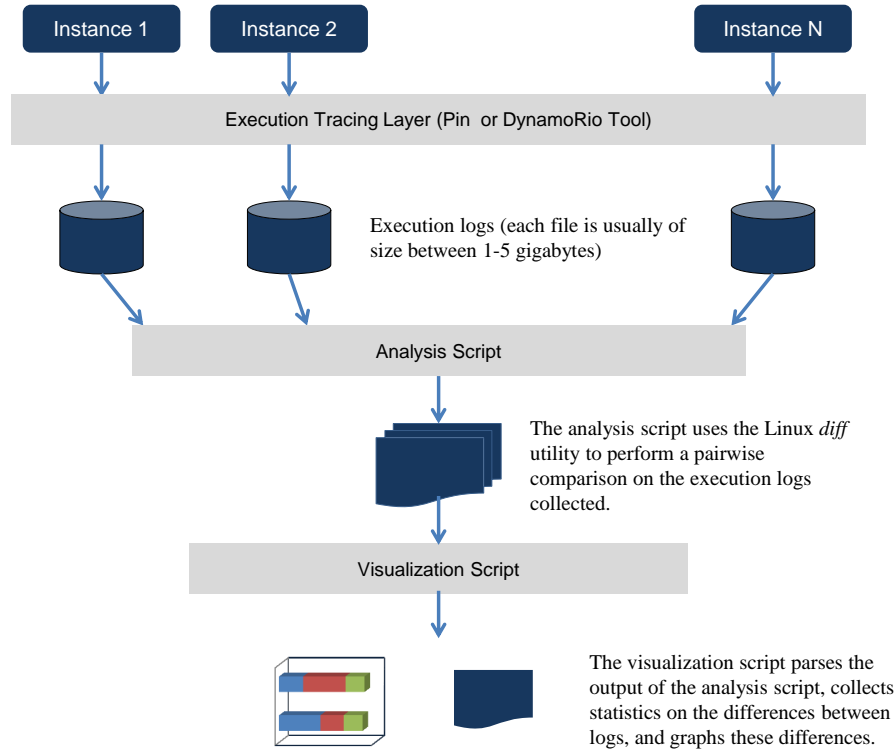


Figure 2-4: Steps involved in measuring execution nondeterminism.

Pin and DynamoRio are runtime frameworks that enable inspection and arbitrary transformation of user-mode application code as it executes. We used both Pin and DynamoRio to study the behavior of Linux services in order to verify the accuracy of our results. However, we relied on Pin more than DynamoRio because it gets injected

into application code earlier than DynamoRio and thus provides greater instruction coverage for our purpose. Figure 2-4 shows the simple steps involved in collecting data on nondeterminism using dynamic instrumentation. The next section explains each of these steps in detail, using a simple “Hello, world!” program as an example.

2.2.1 Measuring nondeterminism in a simple *C* program

This section outlines the data collection scheme described in Figure 2-4 in detail with the help of an example: the simple “Hello, world!” program outlined in Figure 2-5. For this example, we disabled ASLR (Address Space Layout Randomization) on the Ubuntu VM described in section 2.1.

```
1 #include <stdio.h>
2
3 int
4 main(int argc, char* argv[])
5 {
6     printf(“Hello, world!\n”);
7     return 0;
8 }
```

Figure 2-5: A “Hello, world!” program in C.

Execution Tracing Layer

As shown in Figure 2-4, the first step in data collection involves running the target program a few times across identical VMs. Ideally, these different executions are done concurrently or as close as possible in time to model the boot storm scenario accurately. In our scheme, we wrote a Pin tool that:

1. logs each x86 instruction executed by the target process, along with the new values of any affected registers,
2. records values written to or read from memory,
3. intercepts all signals received, and records the instruction counts corresponding to the timing of any signals, and

4. monitors all system calls made by the target process, and logs any corresponding side-effects to memory or registers.

At present, our Pin tool traces the main process for an application and does not follow any child processes spawned by it for simplicity. This prevents us from including the user-mode instructions executed from child processes in our traces, but we get sufficiently high coverage to get a good understanding of the target program’s behavior. We treat spawned child processes as part of the outside world, and trace their interactions with the original process (e.g. via signals or pipes).

Implementation of the execution tracing layer required a close examination of the Linux system call interface, because we had to identify the side-effects of each system call. Figure 2-6 shows an excerpt from a trace generated by our Pin tool while running the “Hello, World” program. Our tool records every instruction executed in user-space by the process for the “Hello, world” program once Pin gets control; this allows us to include program initialization and library code in our analysis.

__dl_make_stack_executable	0xb7ff6210	pop edx	edx = 0xb7ff1040, esp = 0xbffff29c
Read 0xbffff2d4 = *(UINT32*)0xbffff29c			
__dl_make_stack_executable	0xb7ff6211	mov ecx, [esp]	ecx = 0xbffff2d4
__dl_make_stack_executable	0xb7ff6214	mov [esp], eax	
Write *(UINT32*)0xbffff29c = 0xb6415b90			
Read 0xbffff8f8 = *(UINT32*)0xbffff2a0			
__dl_make_stack_executable	0xb7ff6217	mov eax, [esp+0x4]	eax = 0xbffff8f8
Read 0xb6415b90 = *(UINT32*)0xbffff29c			
__dl_make_stack_executable	0xb7ff621b	ret 0xc	esp = 0xbffff2ac
__libc_start_main	0xb6145b90	push ebp	esp = 0xbffff2a8
Write *(UINT32*)0xbffff2a8 = 0			

mmap2() called			
addr = 0	length = 4096	prot = 3	flags = 34
fd = -1	pgoffset = 0		
ret_val = 0xb62dd000			
mmap2() returned			
write() called			
fd = 1			
pbuf = 0xb62dd000			
count = 14			
bytes written = 14			
buf contents:			
buf[0] = H	buf[1] = e	buf[2] = l	buf[3] = l
buf[4] = o	buf[5] = ,	buf[6] =	buf[7] = w
buf[8] = o	buf[9] = r	buf[10] = l	buf[11] = d
buf[12] = !	buf[13] = .		
write() returned			

Figure 2-6: Excerpts from the log files generated by the execution tracing layer. The top half shows x86 instructions executed in user-space by the “Hello, world!” process, including instruction addresses, limited symbolic information, affected register values and memory addresses. The lower half shows part of the system call log.

Analysis Script

The analysis script uses the Linux *diff* utility to perform pairwise comparisons of the log files generated by multiple executions of the target application. Using the `suppress-common`, `side-by-side` and `minimal` flags, the analysis script produces two output files:

1. A *delta* file that contains only instructions that were either conflicting between the two logs or missing in one log, and
2. A *union* file that contains all instructions executed in the two logs, while distinguishing instructions included in the delta file from others.

<pre> Read 0xcb37 = *(UINT16*)0xbffff41b .text 0xb7fe4dfe movzx edx, word ptr [eax] edx = 0xcb37 Write *(UINT16*)0xbffff10d = 0xcb37 Read 0x49 = *(UINT8*)0xbffff41d .text 0xb7fe4e05 movzx eax, byte ptr [eax+0x2] eax = 0x49 Write *(UINT8*)0xbffff10f = 0x49 Read 0x49cb3700 = *(UINT32*)0xbffff10c </pre>	<pre> Read 0xedf8 = *(UINT16*)0xbffff41b .text 0xb7fe4dfe movzx edx, word ptr [eax] edx = 0xedf8 Write *(UINT16*)0xbffff10d = 0xedf8 Read 0x25 = *(UINT8*)0xbffff41d .text 0xb7fe4e05 movzx eax, byte ptr [eax+0x2] eax = 0x25 Write *(UINT8*)0xbffff10f = 0x25 Read 0x25edf800 = *(UINT32*)0xbffff10c </pre>
<pre> .text 0xb7fe4dea mov dword ptr [ebp-0x4], edi Write *(UINT32*)0xbffff118 = 0xb7fff524 .text 0xb7fe4ded mov dword ptr [ebp-0x10], 0x0 Write *(UINT32*)0xbffff10c = 0 Read 0xbffff41b = *(UINT32*)0xb7fef24 .text 0xb7fe4df4 mov eax, dword ptr [ebx-0xd0] eax = 0xbffff41b .text 0xb7fe4dfa test eax, eax eflags = 0x286 .text 0xb7fe4dfc jz 0xb7fe4e51 </pre>	<pre> .text 0xb7fe4dea mov dword ptr [ebp-0x4], edi Write *(UINT32*)0xbffff118 = 0xb7fff524 .text 0xb7fe4ded mov dword ptr [ebp-0x10], 0x0 Write *(UINT32*)0xbffff10c = 0 Read 0xbffff41b = *(UINT32*)0xb7fef24 .text 0xb7fe4df4 mov eax, dword ptr [ebx-0xd0] eax=0xbffff41b .text 0xb7fe4dfa test eax, eax eflags = 0x286 .text 0xb7fe4dfc jz 0xb7fe4e51 </pre>
<pre> Read 0xcb37 = *(UINT16*)0xbffff41b .text 0xb7fe4dfe movzx edx, word ptr [eax] edx = 0xcb37 Write *(UINT16*)0xbffff10d = 0xcb37 Read 0x49 = *(UINT8*)0xbffff41d .text 0xb7fe4e05 movzx eax, byte ptr [eax+0x2] eax = 0x49 Write *(UINT8*)0xbffff10f = 0x49 Read 0x49cb3700 = *(UINT32*)0xbffff10c </pre>	<pre> Read 0xedf8 = *(UINT16*)0xbffff41b .text 0xb7fe4dfe movzx edx, word ptr [eax] edx = 0xedf8 Write *(UINT16*)0xbffff10d = 0xedf8 Read 0x25 = *(UINT8*)0xbffff41d .text 0xb7fe4e05 movzx eax, byte ptr [eax+0x2] eax = 0x25 Write *(UINT8*)0xbffff10f = 0x25 Read 0x25edf800 = *(UINT32*)0xbffff10c </pre>

Figure 2-7: Excerpts from the diff files generated by the analysis script. The top half shows instructions from the delta file; these all have different side-effects in the two logs (as indicated by the |). The bottom half shows instructions from the union file. Conflicting instructions are highlighted; others are found in both logs.

Figure 2-7 shows an excerpt from the union and delta files generated for the “Hello, world!” program. Given several traces, the delta and union files can be constructed from the two executions that are the most different or have the median difference. The much smaller size of the delta file makes it suitable for diagnosing sources of nondeterminism in an application.

Visualization Script

The visualization script reads the union file to compute statistics on the extent of any differences in the original logs, and generates diagrams to capture the different execution traces of the program.

In particular, it derives three key metrics after processing the union file:

1. *Length of Common Prefix (P)*: This is the number of instructions common to both logs starting from the beginning and up to the point of first divergence.
2. *Longest Common Substring (LS)*: This is the longest sequence of consecutive instructions that are common to both logs.
3. *Longest Common Subsequence (LCS)*: Intuitively, this is the “overlap” in the logs; it is the length of the longest sequence of identical instructions in both logs. Instructions in the LCS must be in the same order in both logs, but they are not required to be adjacent.

For instance, if the first instance of a program executes the instruction sequence $I_1 = [A, B, C, D, E, F]$, and the second instance of the same program executes the instruction sequence $I_2 = [A, B, X, D, E, F, Y]$, then: the common prefix is $[A, B]$; the longest common substring is $[D, E, F]$, and the longest common subsequence is $[A, B, D, E, F]$. In general, the longest common subsequence (LCS) of two traces is arguably the best indicator of the extent of determinism in two executions of a program. The other two metrics are important for evaluating the feasibility of deduplication of execution as a solution to the boot storm problem. In general, we want the common prefix (P) and the longest common substring (LS) of the two logs to be as large as possible to ensure that concurrently booting VMs do not need to branch execution or communicate with each other too quickly (see Chapter 5).

For the “Hello, world!” program, if ASLR is enabled, the two logs have very little overlap ($< 1\%$), and the common prefix and longest common substring are on the order of 10 instructions. With ASLR disabled, one may expect the two traces to look identical (because of the simplicity of the program), but there is still some nondeterminism in the instruction sequences (see Table 2.1 and Figure 2-8).

Table 2.1: Nondeterminism profile of “Hello, world!” program (ASLR disabled)

Common Prefix	21.49 percent
Longest Common Substring	67.70 percent
Longest Common Subsequence	99.98 percent
Conflict Ratio	0.02 percent
Conflicting Instructions	32

Figure 2-8 shows divergences in program execution over time. This representation allows us to visually inspect the union file and figure out the distribution and nature of conflicting instructions. For the “Hello, world!” program, we can see that while divergences were spread out near the beginning and end of the program, they were bursty and short-lived (as indicated by the thin black lines). This is a common trend, even for complex programs such as Linux services, as discussed in Section 2.3.

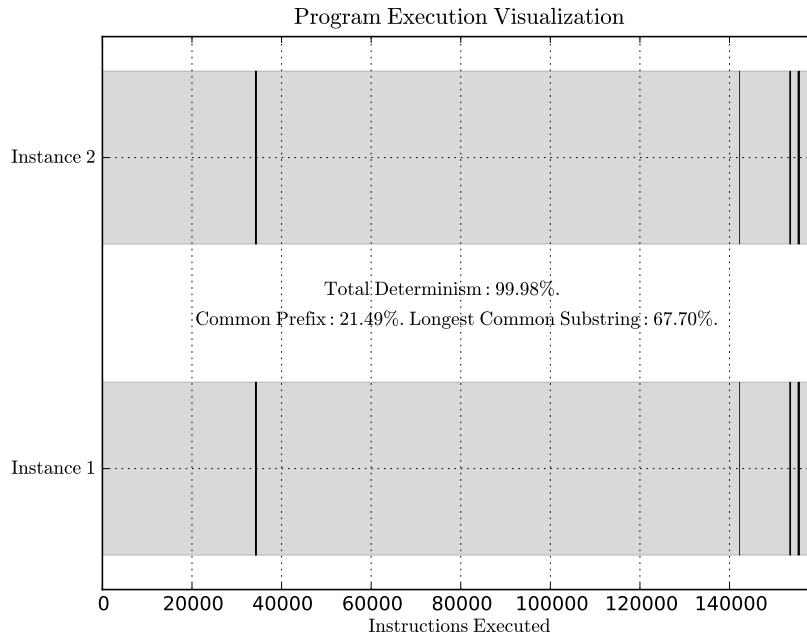


Figure 2-8: Visualization of “Hello, world!” program execution. The thin black lines represent conflicts between the two instances of the program.

2.2.2 Quantifying Nondeterminism

As mentioned in the previous section, we use the common prefix (P), the longest common subsequence (LCS), the longest substring (LS) and the distribution of conflicting instructions in separate instruction streams to measure nondeterminism.

While the conflict ratio measured by our analysis script is usually quite small (e.g. 0.02% for “Hello, world!”), its importance and impact is disproportionately larger. As shown in Figure 2-9, the analysis script ignores the cascade effect and only considers instructions that *originate* or actively *propagate* nondeterminism in calculating the conflict ratio.

<div> <div>mov eax, [edx] \$ eax = 0x141</div> <div> <div>mov ecx, [ebx] \$ ecx = 0x241fb4</div> <div>add ecx, 1 \$ ecx = 0x241fb5</div> </div> <div><N other instructions that do not read/write to eax></div> <div>mov eax, edx \$ eax = 0x1</div> </div>	<div> <div>mov eax, [edx] \$ eax = 0x171</div> <div> <div>mov ecx, [ebx] \$ ecx = 0x241fb4</div> <div>add ecx, 1 \$ ecx = 0x241fb5</div> </div> <div><N other instructions that do not read/write to eax></div> <div>mov eax, edx \$ eax = 0x1</div> </div>
<div> <div>mov eax, [edx] \$ eax = 0x141</div> <div> <div>mov ecx, eax \$ ecx = 0x141</div> <div>add ecx, 1 \$ ecx = 0x142</div> </div> </div>	<div> <div>mov eax, [edx] \$ eax = 0x171</div> <div> <div>mov ecx, eax \$ ecx = 0x171</div> <div>add ecx, 1 \$ ecx = 0x172</div> </div> </div>

Figure 2-9: The top image shows an example of the cascade effect: the red instruction represents a real conflict in `eax`. The light-blue instructions have the same side-effects across the two logs because they do not touch `eax`. The value of `eax` is different in the blue instructions and converges only after it is written by the green instruction. The cascade effect refers to the nondeterministic register state that results in the light-blue instructions because of an earlier conflict, even though the instructions themselves are not reading or writing any nondeterministic values. If we included the cascade effect, the measured conflict ratio in this trace excerpt is $(N + 3)/(N + 4)$ instead of the $1/(N + 4)$ we will report.

The bottom image shows an example of the propagation effect: the red instruction again represents a conflict in `eax`. The light-blue instructions do not generate any nondeterminism themselves, but they have conflicting side-effects because they read `eax`. In this case, we report a conflict ratio of 1.

Since it ignores the cascade effect and includes the propagation effect, we effectively simulate a form of *taint* analysis on register and memory contents to measure the true impact of any nondeterminism in a program. Our approach automatically groups instructions that generate and propagate nondeterminism in the delta files, making it easier for us to diagnose the sources of nondeterminism.

One element missing from our study of nondeterminism is that we do not account for timing-related nondeterminism directly. For instance, two programs that execute precisely the same set of instructions but take different amounts of time doing so (e.g. due to variable latency of blocking system calls) are fully deterministic according to our definition. We deliberately exclude timing considerations because it is acceptable for some VMs to lag behind others in the boot storm scenario, as long as the same instructions are executed. When timing-related nondeterminism affects program execution e.g. through differences in signal delivery, I/O ordering or time-related system calls (see Chapter 3), it automatically gets factored in our analysis.

2.3 Results for Linux services

Table 2.1 shows the results from applying our data collection scheme on a set of Linux services and daemons that are typically launched at boot.

We can immediately see that:

1. The common prefix (P) in our sample of Linux services is on average about 3%, which is quite small and indicates that nondeterminism typically surfaces relatively early in program execution.
2. The longest substring (LS), usually close to 25%, is substantially larger than the common prefix (P). This shows that execution typically does not permanently diverge after the initial differences.
3. The longest common subsequence (LCS) or general determinism is in general much higher – about 90% on average – which indicates that a large majority of instructions in the Linux services overlap across different executions.

Table 2.2: Nondeterminism profile of Linux services and daemons (ASLR disabled)

Application	Prefix (P)	Longest Substring (LS)	Determinism (LCS)
ntp , 14 loop iterations	11.65%	22.08%	89.21%
cron , 30 loop iterations	1.58%	53.21%	98.38%
cups , 10 loop iterations	2.45%	25.20%	94.25%
daemon A , i loop iterations	$p\%$	$ls\%$	$lcs\%$
daemon B , i loop iterations	$p\%$	$ls\%$	$lcs\%$
daemon C , i loop iterations	$p\%$	$ls\%$	$lcs\%$
daemon D , i loop iterations	$p\%$	$ls\%$	$lcs\%$
daemon E , i loop iterations	$p\%$	$ls\%$	$lcs\%$
daemon F , i loop iterations	$p\%$	$ls\%$	$lcs\%$
daemon G , i loop iterations	$p\%$	$ls\%$	$lcs\%$
Aggregate	$x\%$	$y\%$	$z\%$

Given the discussion in Section 2.2.2, a conflict ratio of about 10% on average hints that there is a non-trivial amount of nondeterminism in our sample programs, despite a very high average LCS. The distribution of the 10% conflicting instructions is surprisingly similar across different programs.

Figure 2-10, an execution profile of **ntp**, is representative of most execution traces. Generally, conflicting instructions are spread throughout the program execution, but tend to occur more frequently towards the end. Nondeterminism does not seem to cause permanent execution divergences, even though there is nontrivial amount of control-flow divergence in some programs. In fact, execution seems to diverge and reconverge very frequently. The execution profile of **cron** is somewhat unique because it has a higher LCS and LS than other traces. It is difficult to reconcile the low measured conflict ratio for **cron** (less than 2%), with the higher conflict ratio visually suggested by Figure 2-11. Figure 2-12 explains this discrepancy: it shows that while the absolute number of conflicting instructions is small, these conflicts occur in bursts and visually group together. While the bursty nature of nondeterminism is particularly prominent in Figure 2-12, it is common to all the services we profiled. Table 2.3 shows that the longest control flow divergence or the longest string of consecutive conflicts is typically very small (i.e. $\ll 1\%$) in our sample programs.

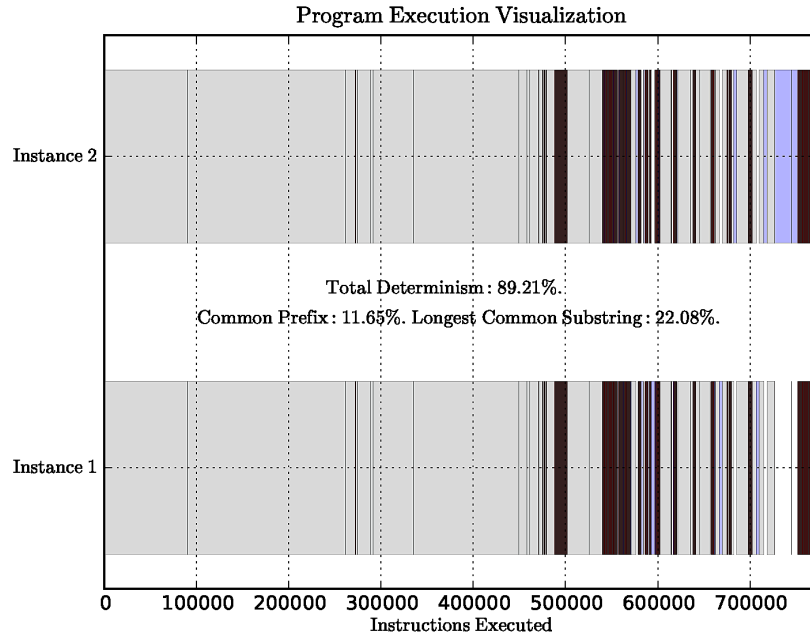


Figure 2-10: Visualization of `ntp` program execution. The thin black lines represent conflicts between the two instances of the program, whereas the thin blue lines represent control flow divergences.

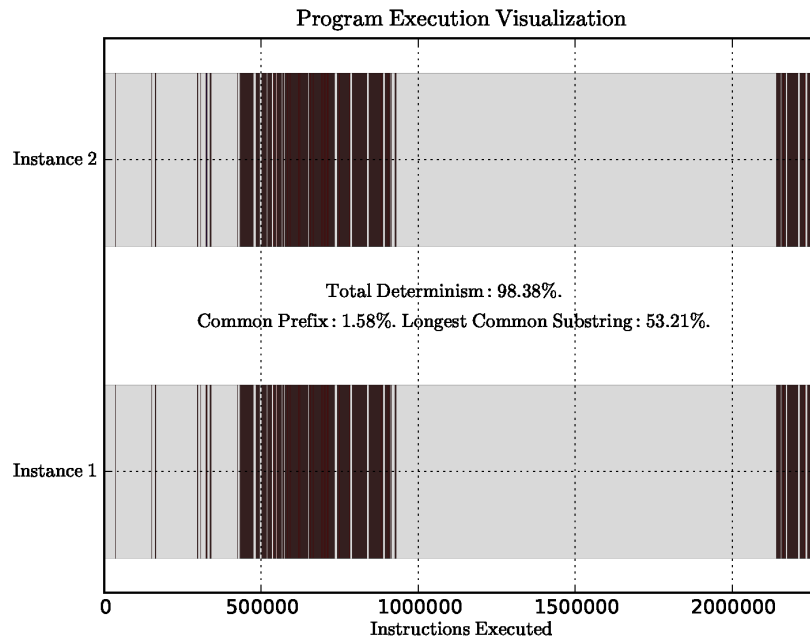


Figure 2-11: Visualization of `cron` program execution. The thin black lines represent conflicts between the two instances of the program.

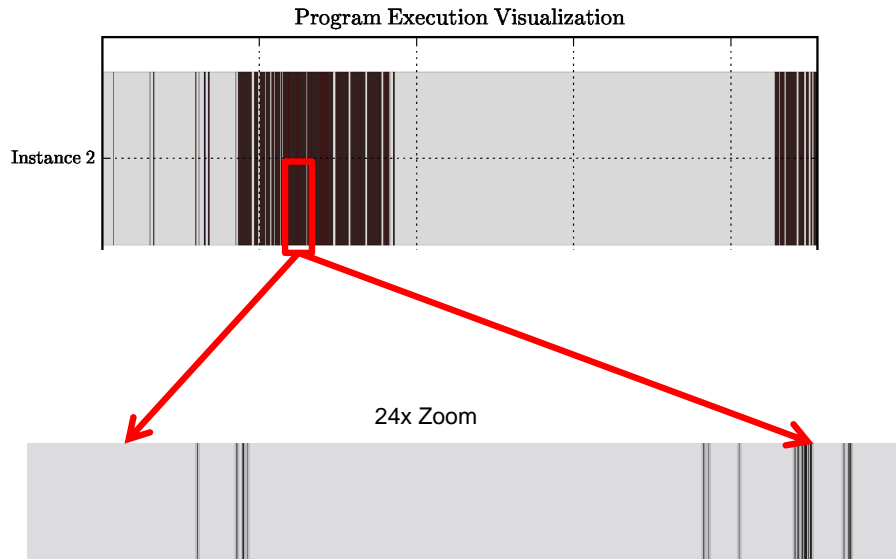


Figure 2-12: Looking closely at the `cron` program execution reveals that conflicts occur in short bursts.

Table 2.3: Measuring burstiness of nondeterminism in Linux services

Application	Max. Consecutive Conflicts	Max. Control Flow Divergence
<code>ntp</code> , 14 loop iterations	0.03%	2%
<code>cron</code> , 30 loop iterations	0.08%	0.003%
<code>cups</code> , 10 loop iterations	$c\%$	$c\%$
daemon A, i loop iterations	$p\%$	$ls\%$
daemon B, i loop iterations	$p\%$	$ls\%$
daemon C, i loop iterations	$p\%$	$ls\%$
daemon D, i loop iterations	$p\%$	$ls\%$
daemon E, i loop iterations	$p\%$	$ls\%$
daemon F, i loop iterations	$p\%$	$ls\%$
daemon G, i loop iterations	$p\%$	$ls\%$
Aggregate	$x\%$	$y\%$

2.4 Summary

This chapter presented a brief overview of the Linux boot process, and demonstrated our methodology for both quantifying and measuring nondeterminism in programs using dynamic instrumentation. By analyzing user-mode instructions executed by Linux boot services and daemons, we offered evidence that Linux services execute highly overlapping instruction sequences across different runs. We also showed that any conflicts or nondeterminism in such services occurs in bursts; nondeterminism does not cause executions to permanently diverge; divergence and convergence occur very quickly and repeatedly in our traces.

Chapters 3 and 4 will offer insight into the sources of nondeterminism behind these statistics. Chapter 5 will look at the implications of our results for a possible solution to the bootstorm problem.

Chapter 3

Overcoming Nondeterminism in Linux services

This chapter summarizes the sources of nondeterminism in Linux services discovered through our experiments (Section 3.1). It also explains the techniques we used to simulate deterministic execution (Section 3.2). Finally, it briefly discusses some of the inherent limitations of deterministic execution (Section 3.3).

3.1 Sources of Nondeterminism

In this section, we describe the sources of nondeterminism discovered using the data collection scheme described in Chapter 2. This study of nondeterminism reveals subtle interactions between user-mode applications, commonly used system libraries (e.g. the `libc` library), the Linux operating system and the external world. While our results are derived from analyzing a small set of complex programs, they include all sources of application-level nondeterminism that have been described in literature. Unlike existing work, however, we cover the various interfaces between user-mode programs and the Linux kernel in considerable detail.

3.1.1 Linux Security Features

Address Space Layout Randomization (ASLR)

Address Space Layout Randomization (ASLR) involves random arrangement of key memory segments of an executing program. When ASLR is enabled, virtual addresses for the base executable, shared libraries, the heap, and the stack are different across multiple executions. ASLR hinders several kinds of security attacks in which attackers have to predict program addresses in order to redirect execution (e.g. *return-to-libc* attacks). As mentioned earlier, two execution traces of even a simple program in *C* are almost entirely different when ASLR is enabled because of different instruction and memory addresses.

Canary Values and Stack Protection

Copying a *canary* – a dynamically chosen global value – onto the stack before each function call can help detect buffer overflow attacks, because an attack that overwrites the return address will also overwrite a copy of the canary. Before a `ret`, a simple comparison of the global (and unchanged) canary value with the (possibly changed) stack copy can prevent a buffer overflow attack.

In 32-bit Linux distributions, the *C* runtime library, `libc`, provides a canary value in `gs:0x14`. If Stack Smashing Protection (SSP) is enabled on compilation, `gcc` generates instructions that use the canary value in `gs:0x14` to detect buffer overflow attacks. Because Pin gets control of the application before `libc` initializes `gs:0x14`, multiple execution traces of a program will diverge when `gs:0x14` is initialized and subsequently read. The manner in which the canary value in `gs:0x14` is initialized depends on the `libc` version. If randomization is disabled, `libc` will store a fixed terminator canary value in `gs:0x14`; this does not lead to any nondeterminism. When randomization is enabled, however, some versions of `libc` store an unpredictable value in `gs:0x14` by reading from `‘/dev/urandom’` or by using the `AT_RANDOM` bytes provided by the kernel (see Section 3.1.2).

Pointer Encryption

Many stateless APIs return data pointers to clients that the clients are supposed to supply as arguments to subsequent function calls. For instance, the `setjmp` and `longjmp` functions can be used to implement a try-catch block in *C*: `setjmp` uses a caller-provided, platform-specific `jmp_buf` structure to store important register state that `longjmp` later reads to simulate a return from `setjmp`. Since the `jmp_buf` instance is accessible to clients of `setjmp` and `longjmp`, it is possible that the clients may advertently or inadvertently overwrite the return address stored in it and simulate a buffer-overflow attack when `longjmp` is called.

Simple encryption schemes can detect mangled data structures. For instance, in 32-bit Linux, `libc` provides a *pointer guard* in `gs:0x18`. The idea behind the pointer guard is the following: to encrypt a sensitive address p , a program can compute $s = p \oplus \text{gs:0x18}$, optionally add some bit rotations, and store it in a structure that gets passed around. Decryption can simply invert any bit rotations, and then compute $p = s \oplus \text{gs:0x18}$ back. Any blunt writes to the structure from clients will be detected because decryption will likely not produce a valid pointer. Pointer encryption is a useful security feature for some APIs and is used by some versions of `libc` to protect addresses stored in `jmp_buf` structures.

The `libc` pointer guard has different values across multiple runs of a program, just like the canary value. Initialization of the `libc` pointer guard can therefore be a source of nondeterminism in program execution. In some versions of `libc`, the value of `gs:0x18` is the same as the value of `gs:0x14` (the canary). In others, the value of `gs:0x18` is computed by XORing `gs:0x14` with a random word (e.g. the return value of the `rdtsc` x86 instruction), or reading other `AT_RANDOM` bytes provided by the kernel (Section 3.1.2).

3.1.2 Randomization Schemes

As already clear from Section 3.1.1, randomization schemes can lead to significant nondeterminism in programs. Applications generally employ pseudorandom number generators (PRNGs), so they need only a few random bytes to *seed* PRNGs.

These few random bytes are usually read from one of few popular sources:

- *The ‘/dev/urandom’ special file.* Linux allows running processes to access a random number generator through this special file. The entropy generated from environmental noise (including device drivers) is used in some implementations of the kernel random number generator.
- *AT_RANDOM bytes.* Using `open`, `read` and `close` system-calls to read only a few random bytes from ‘/dev/urandom’ can be computationally expensive. To remedy this, some recent versions of the Linux kernel supply a few random bytes to all executing programs through the `AT_RANDOM` auxiliary vector. ELF auxiliary vectors are pushed on the program stack before a program starts executing below command-line arguments and environmental variables.
- *The `rdtsc` instruction.* The `rdtsc` instruction provides an approximate number of ticks since the computer was last reset, which is stored in a 64-bit register present on x86 processors. Computing the difference between two successive calls to `rdtsc` can be used for timing whereas a single value returned from `rdtsc` lacks any useful context. The instruction has low-overhead, which makes it suitable for generating a random value instead of reading from ‘/dev/urandom’.
- *The current time or process ID.* System calls that return the current process ID (Section 3.1.3) or time (Section 3.1.4) generate unpredictable values across executions, and are commonly used to seed PRNGs.
- *Miscellaneous:* There are several creative ways to seed PRNGs, including using *www.random.org* or system-wide performance statistics. Thankfully, we have not observed them in our analysis of Linux services.

Randomization-related nondeterminism thus usually originates from any external sources used to seed PRNGs; if the seeds are different across multiple executions, PRNGs further propagate this nondeterminism.

3.1.3 Process Identification Layer

In the absence of a deterministic operating system layer, process IDs for programs are generally not predictable. For instance, a nondeterministic scheduler (Section 3.1.9) could lead to several possible process creation sequences and process ID assignments when a VM boots up.

Given the unpredictability of process IDs, system calls that directly or indirectly interact with the process identification layer can cause divergences across distinct executions of the same program. For instance, system calls that return a process ID e.g. `getpid` (get process ID), `getppid` (get parent process ID), `fork/clone` (create a child process), `wait` (wait for a child process to terminate) return conflicting values across distinct executions. System calls that take process IDs directly as arguments such as `kill` (send a signal to a specific process), `waitpid` (wait for a specific child process to terminate) can similarly propagate any nondeterminism. In fact, `libc` stores a copy of the current process ID in `gs:0x48`, so reads from this address also propagate execution differences.

Apart from system calls, there are other interfaces between the Linux kernel and executing user-mode programs where process IDs also show up:

- *Signals*: If a process registers a signal handler with the `SA_SIGINFO` bit set, then the second argument passed to the signal handler when a signal occurs is of type `siginfo_t*`. The member `siginfo_t.si_pid` will be set if another process sent the signal to the original process (Section 3.1.8).
- *Kernel messages*: The Linux kernel will sometimes use process IDs to indicate the intended recipients of its messages. For instance, `Netlink` is a socket-like mechanism for inter process communications (IPC) between the kernel and user-space processes. `Netlink` can be used to pass networking information between kernel and user-space, and some of its APIs use process IDs to identify communication end-points (Section 3.1.6).

Nondeterminism arising from the unpredictability of process IDs can be further propagated when an application uses process IDs to seed PRNGs (Section 3.1.2), access the `‘/proc/[PID]’` directory (Section 3.1.10), name application-specific files (e.g. `‘myapp-[pid].log’`) or log some information to files (e.g. `‘process [pid] started at [04:23]’`) (Section 3.1.5).

3.1.4 Time

Concurrent runs of the same program will typically execute instructions at (slightly) different times. Clearly, any interactions of a program with timestamps can cause nondeterminism. For instance:

- The `time`, `gettimeofday` and `clock_gettime` system calls return the current time.
- The `times` or `getrusage` system calls return process and CPU time statistics respectively.
- The `adjtimex` system call is used by clock synchronization programs (e.g. `ntp`) and returns a kernel timestamp indirectly via a `timex` structure.
- Programs can access the hardware clock through `‘/dev/rtc’` and read the current time through the `RTC_RD_TIME` `ioctl` operation.
- Many system calls that specify a timeout for some action (e.g. `select`, `sleep` or `alarm`) inform the caller of any unused time from the timeout interval if they return prematurely.
- The `stat` family of system calls returns file modification timestamps; also, many application files typically contain timestamps; network protocols use headers with timestamps as well (Sections 3.1.5 and 3.1.6).

Apart from nondeterminism arising from timestamps, *timing* differences can arise between distinct executions because of variable system-call latencies or unpredictable timing of external events relative to program execution (Sections 3.1.8 and 3.1.7).

3.1.5 File I/O

File contents

If two executions of the same program read different file contents (e.g. cache files), then there will naturally be execution divergence. For concurrently executing Linux services, differences in file contents typically arise from process IDs (Section 3.1.3) or timestamps (Section 3.1.4) rather than semantic differences. Once those factors are controlled, file contents rarely differ.

File Modification Times

Apart from minor differences in file contents, nondeterminism can arise from distinct file modification (**mtime**), access (**atime**) or status-change (**ctime**) timestamps. The **stat** system call is usually made for almost every file opened by a program; the time values written by the system call invariably conflict between any two executions. Most of the time, these timestamps are not read by programs, so there is little propagation. On occasion, however, a program will use these timestamps to determine whether a file is more recent than another, or whether a file has changed since it was last read.

File Size

When a program wishes to open a file in append-mode, it uses **lseek** with **SEEK_END** to move the file cursor to the end, before any **writes** take place. The return value of **lseek** is the updated cursor byte-offset into the file. Clearly, if the length of a file is different across multiple executions of a program, then **lseek** will return conflicting values. Many Linux services maintain log files which can have different lengths due to conflicts in an earlier execution; **lseek** further propagates them. To overcome such nondeterminism, older log files must be identical at the beginning of program execution and other factors that cause nondeterminism must be controlled.

3.1.6 Network I/O

Network Configuration Files

The `libc` network initialization code loads several configuration files into memory (e.g. `‘/etc/resolv.conf’`). Differences in the content, timestamps or lengths of such configuration files can clearly cause nondeterminism. Background daemons (e.g. `dhclient` for `‘/etc/resolv.conf’`) usually update these files periodically in the background. Calls to `libc` functions such as `getaddrinfo` use `stat` to determine if relevant configuration files (e.g. `‘/etc/gai.conf’`) have been modified since they were last read. In our experiments, typically the file modification timestamps – and not the actual contents – of these configuration files vary between different executions.

DNS Resolution

In our experiments, IP addresses are resolved identically by concurrently executing services. However, if DNS-based load-balancing schemes are used, the same server can appear to have different IP addresses.

Socket reads

Bytes read from sockets can differ between executions for a variety of reasons. For instance, different timestamps in protocol headers, or different requests/responses from the external world would be reflected in conflicting socket `reads`. By studying application behavior, it is possible to distinguish between these different scenarios and identify the seriousness of any differences in the bytes read.

In our experiments, we observed nondeterminism in `reads` from `Netlink` sockets. As mentioned in Section 3.1.3, `Netlink` sockets provide a mechanism for inter-process communications (IPC) between the kernel and user-space processes. This mechanism can be used to pass networking information between kernel and user space. `Netlink` sockets use process IDs to identify communication endpoints, which can differ between executions (Section 3.1.3). Similarly, some implementations of `libc` use timestamps to assign monotonically increasing sequence numbers to `Netlink` packets (Section

3.1.4). Nondeterminism can also arise from sockets of the `NETLINK_ROUTE` family, which receive routing and link updates from the kernel; `libc` receives `RTM_NEWLINK` messages when new link interfaces in the computer are detected. When an interface gets discovered or reported, the kernel supplies interface statistics to `libc` such as packets sent, dropped or received. These statistics will obviously vary across different program instances.

Ephemeral Ports

A TCP/IPv4 connection consists of two end-points; each end-point consists of an IP address and a port number. An established client-server connection can be thought of as the 4-tuple (server_IP, server_port, client_IP, client_port). Usually three of these four are readily known: a client must use its own IP, and the pair (server_IP, server_port) is fixed. What is not immediately evident is that the client-side of the connection uses a port number. Unless a client program explicitly requests a specific port number, an *ephemeral port* is used. Ephemeral ports are temporary ports that are assigned from a dedicated range by the machine IP stack. An ephemeral port can be recycled when a connection is terminated. Since the underlying operating system is not deterministic, ephemeral port numbers used by Linux services tend to be different across multiple runs.

3.1.7 Scalable I/O Schemes

Polling Engines

Complex programs like Linux services have many file descriptors open at a given time. Apart from regular files, these special file descriptors could correspond to:

- *Pipes*: Pipes are used for one-way interprocess communication (IPC). Many Linux services spawn child processes; these child processes communicate with the main process (e.g. for status updates) through pipes.
- A *listener socket*: If the program is a server, this is the socket that accepts incoming connections.

- *Client-handler sockets*: If this program is a server, new requests from already connected clients would arrive through these sockets.
- *Outgoing sockets*: If the program is a client for other servers, it would use these sockets to send requests to them.

The classic paradigm for implementing server programs is *one thread or process per client* because I/O operations are traditionally blocking in nature. This approach scales poorly as the number of clients – or equivalently, the number of open special file descriptors – increases. As an alternative, event-based I/O is increasingly used by scalable network applications. In such designs, the main event-thread specifies a set of file descriptors it cares about, and then waits for “readiness” notifications from the operating system on any of these file descriptors by using a system call such as `epoll`, `poll`, `select` or `kqueue`. For instance, a client socket would be ready for reading if new data was received from a client, and an outgoing socket would be ready for writing if an output buffer was flushed out or if the connection request was accepted. The event-thread invokes an I/O handler on each received event, and then repeats the loop to process the next set of events. This approach is often used for design simplicity because it reduces the threads or processes needed by an application; recent kernel implementations (e.g. `epoll`) are also efficient because they return the set of file descriptors that are ready for I/O, preventing the need for the application to iterate through all its open file descriptors.

Event-based I/O can be a source of nondeterminism in programs because the timing of I/O events with respect to each other can be different across multiple executions. Even if I/O events are received in the same order, the same amount of data may not be available from ready file descriptors. Furthermore, when a timeout interval is specified by the application for polling file descriptors, `select` may be completed or interrupted prematurely. In that case, `select` returns the remaining time interval, which can differ between executions (Section 3.1.4).

Asynchronous I/O Systems

Asynchronous I/O APIs (e.g. the Kernel Asynchronous I/O interface in some Linux distributions) allow even a single application thread to overlap I/O operations with other processing tasks. A thread can request an I/O operation (e.g. `aio_read`), and later query the operating system for its status or ask to be notified when the I/O operation has been completed (e.g. `aio_return`). While such APIs are in limited usage, they introduce nondeterminism because of the variable latency and unpredictable relative timing of I/O events.

3.1.8 Signals

A signal is an event generated by Linux in response to some condition, which may cause a process to take an action in response. Signals can be generated by error conditions (e.g. memory segment violations), terminal interrupts (e.g. from the shell), inter-process communication (e.g. parent sends `kill` to child process), or scheduled alarms. Processes register handlers (or function callbacks) for specific signals of interest in order to respond to them.

Signals are clearly external to instructions executed by a single process, as such, they create nondeterminism much the same way as asynchronous I/O: signals can be delivered to multiple executions of the same program in different order; even if signals are received in the same order between different executions, they can be received at different times into the execution of a program.

3.1.9 Concurrency

Multiple possible instruction-level interleavings of threads within a single program, or of different processes within a single operating system are undoubtedly significant sources of nondeterminism in programs. Nondeterminism due to multi-threading has been extensively documented and can cause significant control flow differences across different executions of the same program.

Nondeterminism in the system scheduler is external to program execution, and manifests itself in different timing or ordering of inter-process communication e.g. through pipes (Section 3.1.7), signals (Section 3.1.8), or values written to shared files or logs (Section 3.1.5).

3.1.10 *Procfs*: The ‘/proc/’ directory

Instead of relying on system-calls, user-space programs can access kernel data much more easily using *procfs*, a hierarchical directory mounted at ‘/proc/’. This directory is an interface to kernel data and system information that would otherwise be available via system calls (if at all); thus, many of the sources of nondeterminism already described can be propagated through it.

For instance, ‘/proc/uptime’ contains time statistics about how long the system has been running; ‘/proc/meminfo’ contains statistics about kernel memory management; ‘/proc/net/’ contains statistics and information for system network interfaces; ‘/proc/diskstats/’ contains statistics about any attached disks. These files will differ across multiple executions of a program because of nondeterminism in the underlying operating system.

Apart from accessing system-wide information, a process can access information about its open file descriptors through ‘/proc/[PID]/fdinfo’ (e.g. cursor offsets and status). Similarly, ‘/proc/[PID]/status’ contains process-specific and highly unpredictable statistics, e.g. number of involuntary context switches, memory usage, and parent process ID. Performing a `stat` on files in ‘/proc/[PID]/’ can reveal the process creation time.

3.1.11 Architecture Specific Instructions

Architecture specific instructions such as `rdtsc` and `cpuid` can return different results across program executions. As mentioned before (Section 3.1.2), the `rdtsc` instruction provides the number of ticks since the computer was last reset, which will differ across executions. The `cpuid` instruction can return conflicting hardware information too.

3.2 Simulating Deterministic Execution

3.3 Limitations of Deterministic Execution

This section describes some of the drawbacks of deterministic execution.

Security

To achieve deterministic execution, we have to disable ASLR. We also have to fix canary (`gs:0x14`) or pointer guard (`gs:0x18`) values across many different VMs. Disabling ASLR increases the vulnerability of applications to external attacks. Though canary and pointer guard values are still dynamically chosen in our brand of deterministic execution, they must agree across all VMs. Thus, an adversary who can compromise one VM by guessing its canary could easily attack the other VMs. The fact that we can choose different canary or guard values between different successive bootstorms is some consolation and provides some security.

Randomization

Randomization can be essential for security (e.g. random values may be used to generate keys or certificates), performance, and sometimes simply correctness (e.g. clients may choose random IDs for themselves). Making PRNG seeds agree across all VM instances can entail a compromise on all of these fronts. Thankfully, we have not yet discovered any such issues in the Linux services. Technically, our approach simulates the extremely unlikely – yet possible – scenario that all concurrently executing instances somehow generated the same seeds from external sources.

Time and Correctness

Any programs that rely on precise measurements of time (e.g. through `rdtsc`) will lose correctness. Some Linux services such as `ntp` do need to measure time accurately in order to synchronize the system clock. Our semantics can cause such services to behave incorrectly at start up, because we may give incorrect values of time to `ntp`.

Thankfully, this is not a huge correctness problem because network clock synchronization programs are self-healing and because we ultimately do provide monotonically increasing time values. After the booting process is over, and all VMs branch in execution, `ntp` will synchronize the current time correctly.

I/O aggregation in Network

As indicated earlier, when file contents or bytes read over sockets differ, these could be because of synthetic differences (e.g. timestamps in headers), or because of semantic differences (e.g. different requests or data). We can study application code and use dynamic instrumentation to reconstruct file contents or network packets and overcome synthetic nondeterminism from such sources. In our experiments, we used this approach for `Netlink` packets and application files, but generalizing it to all network sockets and protocols, while possible, would clearly complicate the design of the dynamic instrumentation layer. For semantic differences in I/O, execution would have to branch out.

One possible approach for fixing nondeterminism from external socket `reads` would be to forcibly conform these reads to be identical by replaying them. This approach would work for many Linux services and would simulate the possibility that these services received responses from an external source at the exact same time containing precisely the same data. However, these semantics can also be problematic in terms of correctness for other services e.g. if a network response essentially says *“you have the lock”* or *“your print job was successfully queued”* or *“your client id is 134”*, this approach falls apart.

3.4 Summary

This chapter described sources of non-determinism in Linux services and ways to overcome them in detail. It also discussed some of the limitations of deterministic execution. Chapter ?? describes the effectiveness of the techniques described in this section by presenting a case study of three Linux services.

Chapter 4

Silhouette Execution

This chapter introduces *silhouette* execution as a possible solution to the bootstorm problem (Section 4.1). It then explains the dynamic instrumentation techniques we used to simulate and improve silhouette execution for concurrently booting VMs (Section 4.2). Finally, it evaluates the effectiveness of silhouette execution in solving the bootstorm problem (Section 4.3).

4.1 What is *Silhouette* execution?

Silhouette execution encapsulates the novel design idea that information recorded from one execution of a program can be used to bypass execution steps of subsequent instances of the same program. Precise knowledge of where program execution can diverge is indispensable to such a design. Silhouette execution is designed extend CPU capabilities much like memory overcommitment is designed to extend the memory resources of host machines. One possible design that employs silhouette execution to save on CPU usage from running the same program multiple times is as follows:

- We run one program instance – the *leader* – before we all others.
- Using information collected from the *leader* (e.g. via dynamic instrumentation techniques), we precisely identify instructions where other instances of the program could diverge. We call these instructions *fork-points*.

- From the leader, we also collect an *execution signature* that summarizes its execution between successive fork-points. For a user-space program, this could include a record of memory operations and system calls.
- We do not execute all instructions from execution instances – or *silhouettes* – that are run after the leader. Instead, each silhouette skips execution to the next fork-point by executing memory operations and system-calls before that fork-point. It then executes the *forking* instruction.
- If a forking instruction in a silhouette has different side-effects from the corresponding instruction in the leader, the silhouette branches execution and continues executing instructions of its own. Otherwise, the silhouette bypasses execution to the next fork-point by executing only memory operations and system calls before the next fork-point.

The description above describes how silhouette execution could be used to save on CPU usage from running the same user-space program across multiple concurrently booting VMs. This design can be further generalized to the execution of entire VMs themselves, but then our definitions of fork-points and execution signatures would have to account nondeterminism that can arise from all software layers inside a VM. We focus on the design of using silhouette execution on user-space programs as a first step: after all, as outlined Section 2.1, concurrently booting VMs can saturate host CPUs when they launch many common user-space processes.

The fundamental – and significant – difference between silhouette execution and record-and-replay approaches is that silhouette execution does not semantically alter the executions of silhouettes by making them blindly following the leader. Instead, silhouettes exploit commonality between multiple concurrent executions to reduce CPU overhead. Silhouettes are faithful executions of a program in user-space that can semantically branch from the leader’s execution at any point because we execute all system calls on behalf of silhouettes rather than simply emulating them like record-and-replay approaches.

Several specific metrics are important in evaluating the feasibility of silhouette execution for the bootstorm scenario:

- Ideally, delaying the first fork-point to be as late into silhouette execution as possible would *guarantee* that we avoid executing many redundant instructions and save on CPU usage.
- More generally, having few fork-points is desirable because it reduces the design complexity and overhead associated with figuring out whether a silhouette has diverged in execution from a leader or not.
- Having fork-points that are separated by very many instructions can help compress execution logs. For instance, we could forget intermediate values of memory locations and only remember their final values instead.
- Finally, programs that have a high ratio of user-mode instructions to system-calls and memory operations would clearly improve the savings from silhouette execution.

4.2 Simulation Scheme

We modified our data collection scheme from Chapter 2 (shown in Figure 2-4) to simulate and evaluate silhouette execution in the bootstorm scenario.

As shown by Figure 4-1, we run one instance of the program – the *leader* – before all others. For the leader, we generate an *execution log*, as before, but we also augment the log by summarizing information about the sources of nondeterminism described in Section 3.1. For instance, we record information about signal timing, process IDs, time-related system calls in the *execution signature* file. Our dynamic instrumentation tool uses the execution signature of the leader to modify the instruction sequences executed by subsequently executions (the *followers*), to boost determinism as much as possible. Even though we use specific terminology, this simulation scheme can be reconciled with multiple possible solutions to the boot storm scenario that involve:

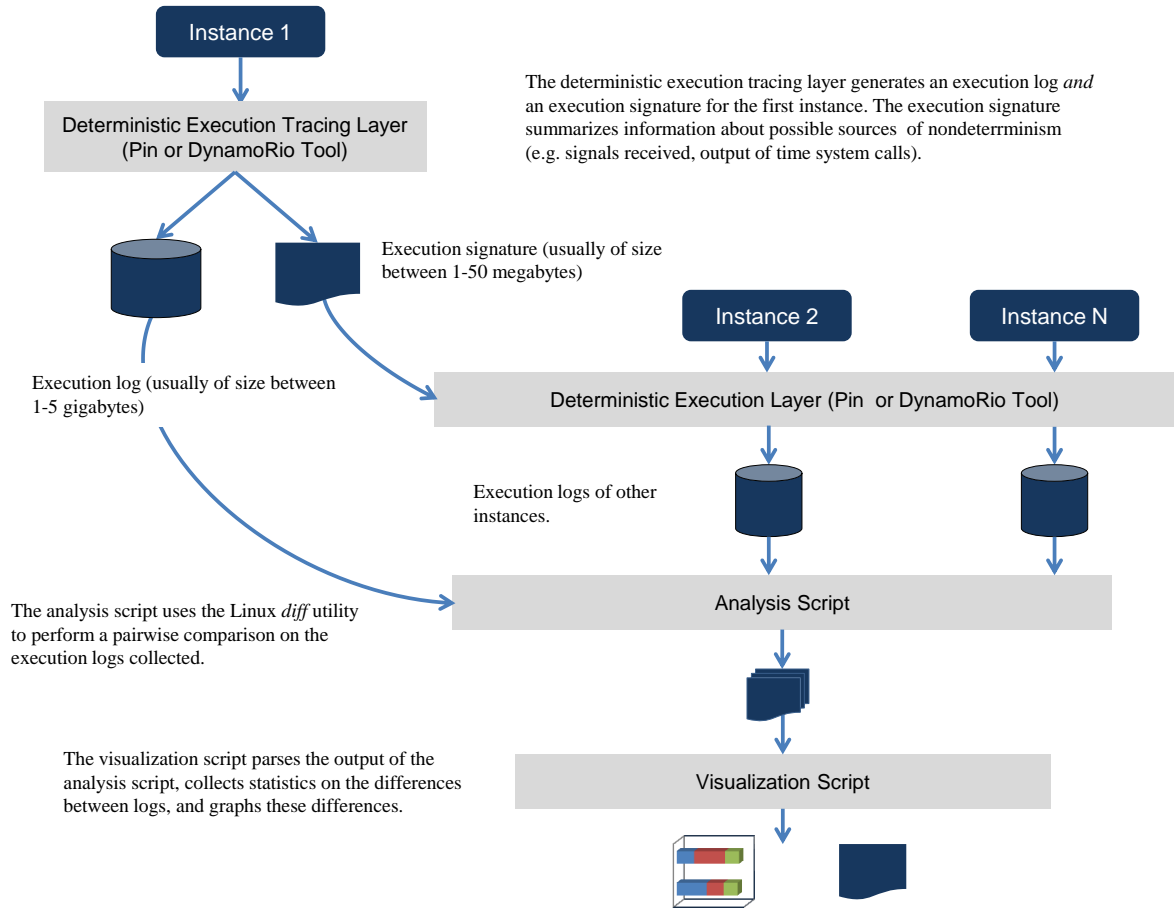


Figure 4-1: Simulation of Silhouette Execution in a bootstorm scenario.

- booting one VM before others, then using its execution trace to *speed-boot* other VMs in a less hardware-intensive manner, or
- running VMs concurrently but with an assigned leader; the VMs communicate with each other to overcome nondeterminism.

In the first case, the execution signature file conceptually represents the trace of the first VM used to speed-boot other VMs. In the second case, it represents a log of the communications sent from the leader sent to concurrently booting VMs. Possible design ideas for the bootstorm scenario are described in Chapter 5.

We now briefly describe how dynamic instrumentation can be used to overcome the sources of nondeterminism described in 3.1.

Address Space Layout Randomization (ASLR)

Existing record-and-replay systems get around ASLR by forcing the operating system to use the same address space layout across different runs. A slightly more complex approach would use base/offset computations to translate two equivalent addresses between different executions. For our experiments, we simply disabled ASLR using the command `sudo kernel.randomize_va_space=0` to simulate the case where we nudge the operating system to construct similar address spaces for the same process.

Canary and Pointer Guard Values

Dynamic instrumentation can be used to force canary (`gs:0x14`) and pointer guard (`gs:0x18`) values to agree across distinct executions of the same program: instructions that initialize them can be modified or replaced; the `AT_RANDOM` bytes provided by the kernel can be modified before they are read by the application; values read from `‘/dev/urandom’` can be intercepted and modified; the `rdtsc` instruction can be emulated. The last three approaches are needed to control nondeterminism from randomization anyway and suffice for our purpose.

Randomization and Time

To overcome nondeterminism resulting from randomization, we need to intercept the standard techniques used by programs to seed PRNGs. As already mentioned, values read from `‘/dev/urandom’` or the `rdtsc` instruction can be easily replaced using the execution signature file for the first instance.

Time-related system calls can be intercepted in the same manner: the timestamps logged in the execution signature file can be used to force agreement between different executions. For most time-related system calls, high-fidelity replay is not necessary: several timestamps generated during program execution are simply ignored (e.g. from `stat` calls), so they can actually be replaced with any fixed value. Many timestamps returned from system calls are only compared to determine “freshness”: they can be replaced with deterministic ordinal values (e.g 0 or 1) that preserve the original

comparison result.

Conceptually, these approaches simulate the possible but highly unlikely case that all the instances of the same program executed time-related system calls at precisely the same times, and several randomly generated values independently agreed.

Signal Delivery

In order to overcome the unpredictable timing and order of signals, we intercept all signals received by an application and ensure they are delivered at precisely the same instruction counts and in the same order as that indicated in the execution signature file.

Unlike record-and-replay systems, we only deliver signals that are *actually* received. Thus, signals that are received earlier than expected are simply delayed or reordered. If, however, a signal is not received at the expected instruction count, our instrumentation tool simply waits or inserts **nops** until the desired signal is received. If a signal simply refuses to appear for a long time, execution must diverge. In our experiments, this final case does not occur as long as other sources of nondeterminism are controlled.

Process IDs

Nondeterminism from process IDs can be controlled by virtualizing the process ID layer, as shown by Figure 4-2.

Using dynamic instrumentation, we can replace real and unpredictable process IDs from kernel space with virtual and unpredictable process IDs in user space. As outlined in Section 3.1.3, all interfaces which use process IDs need to be carefully monitored so that process IDs can be translated back and forth for correctness.

File I/O

Differences in input file contents across executions would inevitably cause execution to diverge, but overcoming nondeterminism arising from time, randomization or process ID system calls is typically sufficient to ensure that file contents rarely differ in Linux

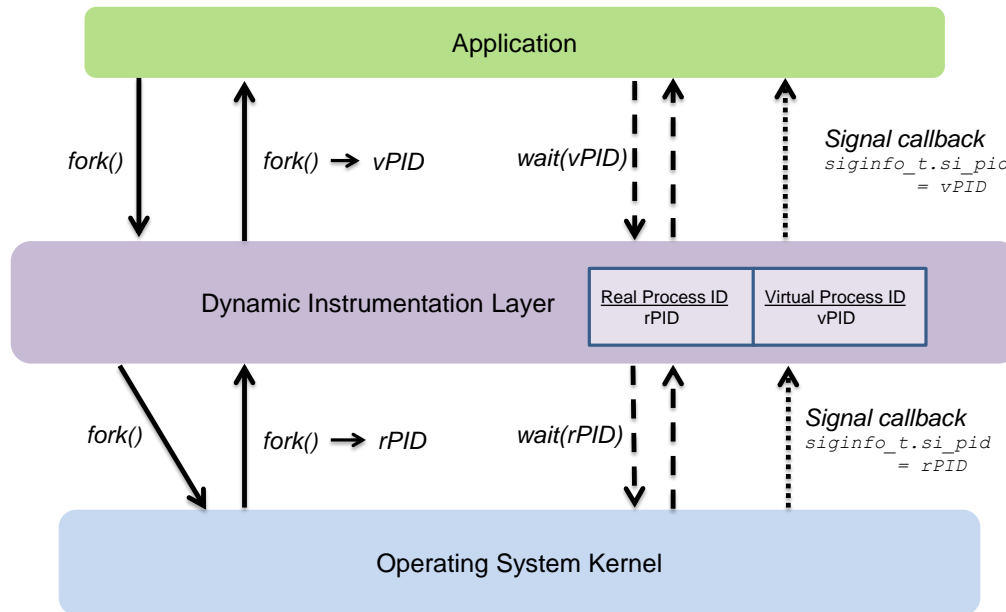


Figure 4-2: All system calls and communications between the Linux user and kernel space are intercepted; the dynamic instrumentation layer uses a PID translation table, and translates between real and virtual process IDs to ensure correctness.

services, if at all. Some files that may differ between two instances on start up (e.g. cache files or logs) can simply be deleted or replaced without sacrificing correctness. Also, as mentioned already, `stat` timestamps are frequently not read, so they can be replaced with fixed constants; when they are read and only compared with other `stat` timestamps, they can be replaced with ordinal numbers that preserve ordering; in all other cases, `stat` system calls can be faithfully replayed.

Network I/O

The content of network configuration files does not change in our experiments, so the strategies described to handle `stat` timestamps are sufficient for them. In the same vein, whenever an address is resolved differently between execution instances because of DNS-based dynamic load balancing, we can intercept and replace resolved IPs with those stored in the execution signature file.

If bytes read from sockets differ across different executions, we need to understand the *context* to determine whether the differences are serious (e.g. due to different requests) or synthetic (e.g. due to timestamps). This can complicate design of the dynamic instrumentation layer because it may have to re-execute application or `libc` logic to understand differences in raw bytes read from system calls. We handle nondeterminism from `Netlink` sockets in this way: the dynamic instrumentation layer re-executes `libc` logic associated with understanding contents of `RTM_NEWLINK` messages to detect nondeterminism from source/destination IDs, sequence numbers or interface statistics. To handle variability in interface statistics, we can simply overwrite them with fixed values. This scheme can be generalized to handle other kinds of network protocols as well.

As an alternative, we could aggressively intercept Linux socket calls and blindly replay them in all followers. When many concurrent executions are reading data from the same network source, this simulates the possibility that all instances see the same results over the network as the first instance. Such an approach, however, can break correctness (Section 3.3).

To overcome nondeterminism from ephemeral ports, we monitor the `bind` or `connect` system calls and change their arguments to explicitly request ports in the ephemeral range rather than let the kernel assign them; if necessary, we can also virtualize ephemeral ports in a similar fashion to how we virtualize process IDs.

Scalable I/O Schemes

To handle nondeterminism caused by unpredictable ordering of I/O events, we use techniques similar to those used for reordering signals, as described by Figure 4-3.

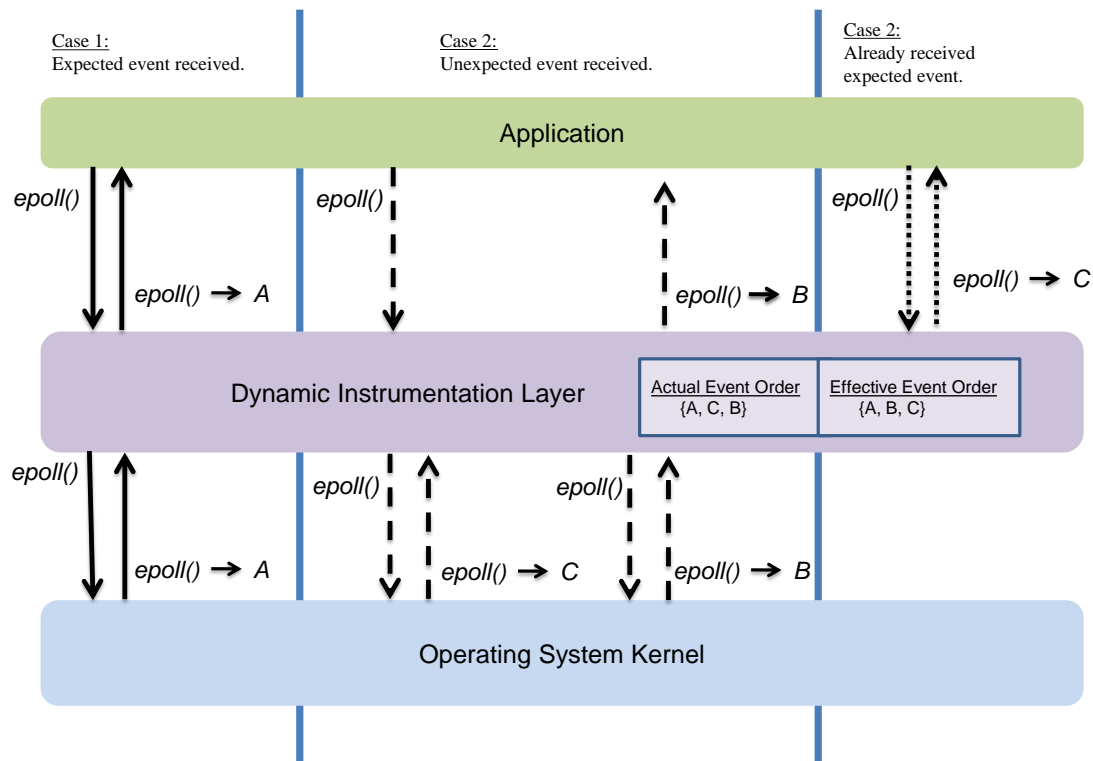


Figure 4-3: We intercept all `epoll` system calls, and use the execution signature file to achieve determinism. We do not “replay” I/O events because only events that actually do occur are delivered to the application instance. This diagram assumes `epoll` returns one event per call for the sake of illustration.

Assuming that `epoll` returns just one event, figure 4-3 illustrates three possible cases that could occur:

- The event returned by a call to `epoll` (A) is the one expected in the execution signature file (A). The instrumentation layer does not modify the system call.

- The desired event (B) has not been received yet, and `epoll` returns an unexpected event (C). The instrumentation layer stores the out-of-order event, and repeatedly calls `epoll` until the expected event is received.
- A call to `epoll` is initiated, and the event desired (C) has already been received. The instrumentation layer does not make a system call and simulates a return from `epoll` with the expected event instead.

Even if I/O events are reordered, it is possible that different amounts of data are available for ready file descriptors across executions. We can mask this effect in the same way we handle signals: if more bytes are available (e.g. through `read`) than expected in the execution signature file, we modify return values and `read` buffers to delay the reading of these bytes until the next `read`. In some corner cases, we may have to “fake” readiness in a call to `epoll`: if all bytes to be read from a file descriptor have been read by the dynamic instrumentation layer (out of which a few have not yet been delivered to the application), there will be no more readiness events even though the application expects them. If less-than-expected bytes are available, we simply wait till they are available by waiting for another readiness update on the same file descriptor inside dynamic instrumentation layer. In our experiments, this approach has been sufficient for overcoming nondeterminism from event-based I/O schemes.

For asynchronous I/O schemes (e.g. `aio_read`), strategies similar to those used for reordering and precisely-timing signals would be necessary to hide variable I/O latency and ordering.

Concurrency

Nondeterminism due to multi-threading has been extensively documented; there is a significant body of work that attempts to overcome such nondeterminism by using deterministic logical clocks or record-and-replay approaches. For our experiments, we did not attempt to enforce a total order on the instructions executed in multi-threaded programs and just measured nondeterminism inside the main process for

each Linux service. To overcome nondeterminism caused by multi-threading, we could incorporate deterministic logical clocks into our design by augmenting the execution signature file.

As mentioned before, a nondeterministic system scheduler can cause variable timing of signals or I/O events, which can be handled using reordering and timing strategies. Work on deterministic operating systems can be extended to overcome this issue in a more systematic manner.

Procfs: The ‘/proc/directory’

Using the techniques described already, I/O operations on *procfs* can be intercepted and modified. We can replay reads from *procfs* using the execution signature file if necessary or replace any statistics with fixed and reasonable values. The dynamic instrumentation layer must respect differences in virtual and real processes: it must modify all `open` system calls with paths of the form ‘`proc/[PID]`’ by switching real and virtual process IDs, and a process must see its parent’s virtual process ID when it reads ‘`/proc/[PID]/status`’.

4.3 Evaluation of Simulation

In order to evaluate the feasibility of this design, we need to

4.4 Summary

Chapter 5

Design Ideas for Deduplication of Execution

5.1 Summary

Chapter 6

Conclusion and Future Work

6.1 Summary

Bibliography

- [1] T. Bergan, N. Hunt, L. Ceze, and S.D. Gribble. Deterministic process groups in dos. *9th OSDI*, 2010.
- [2] D.L. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Citeseer, 2004.
- [3] A.T. Clements, I. Ahmad, M. Vilayannur, and J. Li. Decentralized deduplication in san cluster file systems. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, pages 8–8. USENIX Association, 2009.
- [4] J.G. Hansen and E. Jul. Lithium: virtual machine storage for the cloud. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 15–26. ACM, 2010.
- [5] Solving Boot Storms With High Performance NAS. http://www.storage-switzerland.com/Articles/Entries/2011/1/3_Solving_Boot_Storms_With_High_Performance_NAS.html, 2011. [Accessed 1-August-2011].
- [6] X.F. Liao, H. Li, H. Jin, H.X. Hou, Y. Jiang, and H.K. Liu. Vmstore: Distributed storage system for multiple virtual machines. *SCIENCE CHINA Information Sciences*, 54(6):1104–1118, 2011.
- [7] C.K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices*, volume 40, pages 190–200. ACM, 2005.
- [8] Bootchart: Boot Process Performance Visualization. <http://www.bootchart.org>, 2011. [Accessed 29-July-2011].
- [9] S. Meng, L. Liu, and V. Soundararajan. Tide: achieving self-scaling in virtualized datacenter management middleware. In *Proceedings of the 11th International Middleware Conference Industrial track*, pages 17–22. ACM, 2010.
- [10] VMware Bootstorm on NetApp. <http://ctistrategy.com/2009/11/01/vmware-boot-storm-netapp/>, 2009. [Accessed 29-July-2011].
- [11] M. Olszewski, J. Ansel, and S. Amarasinghe. Scaling deterministic multithreading. *2nd WoDet*, 2011.

- [12] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 2–11. ACM, 2010.
- [13] A. Qureshi, R. Weber, H. Balakrishnan, J. Gutttag, and B. Maggs. Cutting the electric bill for internet-scale systems. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 123–134. ACM, 2009.
- [14] Vijayaraghavan Soundararajan and Jennifer M. Anderson. The impact of management operations on the virtualized datacenter. *SIGARCH Comput. Archit. News*, 38:326–337, June 2010.
- [15] M.W. Stephenson, R. Rangan, E. Yashchin, and E. Van Hensbergen. Statistically regulating program behavior via mainstream computing. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 238–247. ACM, 2010.
- [16] S.B. Vaghani. Virtual machine file system. *ACM SIGOPS Operating Systems Review*, 44(4):57–70, 2010.
- [17] VMware Virtual Desktop Infrastructure. http://www.vmware.com/pdf/virtual_desktop_infrastructure_wp.pdf, 2011. [Accessed 29-July-2011].
- [18] C.A. Waldspurger. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.