Understanding 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

June 2011

Author . . . .
Department of Electrical Engineering and Computer Science
June 21, 2011

Certified by . . . .
Dr. Saman P. Amarasinghe
Professor
Thesis Supervisor

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

Understanding and Controlling Nondeterminism in Linux Services

by

Syed Aunn Hasan Raza

Submitted to the Department of Electrical Engineering and Computer Science
on June 21, 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, they 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 inpsirational and gifted people from the COMMIT group: Marek Olszewski and Qin Zhao. Marek and Qin's expertise and brilliance is probably only eclipsed by their humility and helpfulness. I have learned more from them than I probably realize, and their knowledge of Dynamic Instrumentation and Operating Systems 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 Srini Devadas, Professor Fraans Kaashoek, and Professor Dina Katabi as a Teaching Assistant 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 the never-ending arguments and unproductive 'all-nighters' that accompany our friendship. I also owe a debt of gratitude to my partners-in-crime Prannay Budhraja, 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 in life. 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 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

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

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 new generations of processors support more cores and more 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 desireable 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 minimally follow the leader by executing a tiny subset of the instructions they would otherwise execute; we fork 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 Contrbutions

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

In this document, we:

1. show that nondeterminism in Linux services is bursty and extremely rare;
2. document the sources of non-determinism 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 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 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 specfic application, deterministic execution of programs can be beneficial in many different scenarios in its own right.

The motivations for deterministic multhreading 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: with deterministic execution, a program must execute the exact same instructions across different executions for the same inputs.

Debugging: Erroneous behavior can be more easily reproduced via determininstic 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 non-determinism we discovered in this work. Chapter 4 presents a detailed case study of three Linux services to identify the common context in which non-determinism arises and the strategies that can be used to control it. Chapter 5 presents design ideas for an implementation of deduplication of execution. Chapter 6 summarizes related work. Finally, Chapter 7 concludes this thesis and discusses future work.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 2

Execution Profile of Linux Services

This chapter first 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.

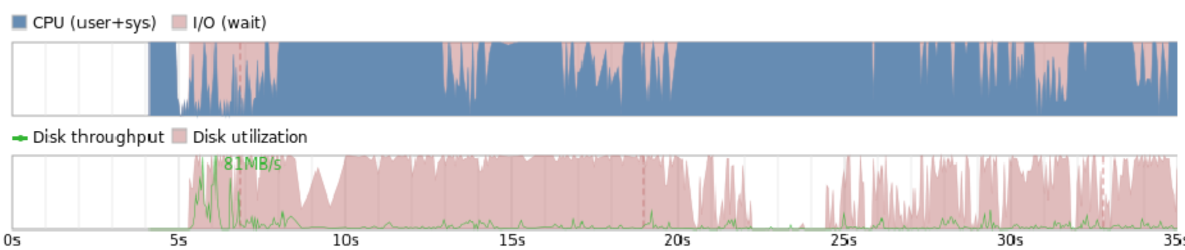

Figure 2-1: CPU and disk activity for a booting Ubuntu VM in the first 35 seconds after `init` is spawned. The first few seconds show no activity because the data collection daemon takes a few seconds to start.

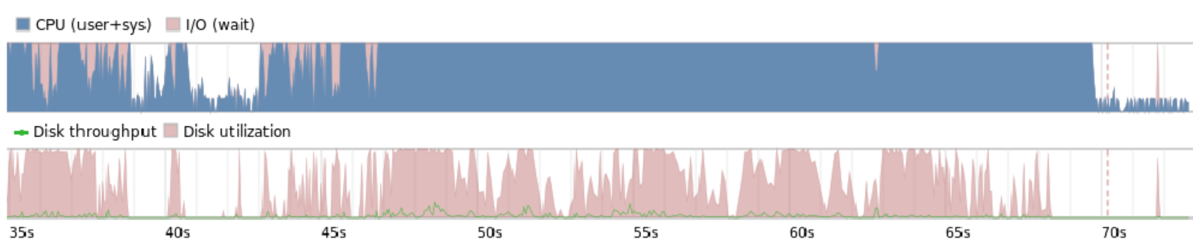

Figure 2-2: (... continued from Figure 2-1) CPU and disk activity for a booting Ubuntu VM 35 seconds after `init` is spawned.

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 dimishes 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 the host 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.

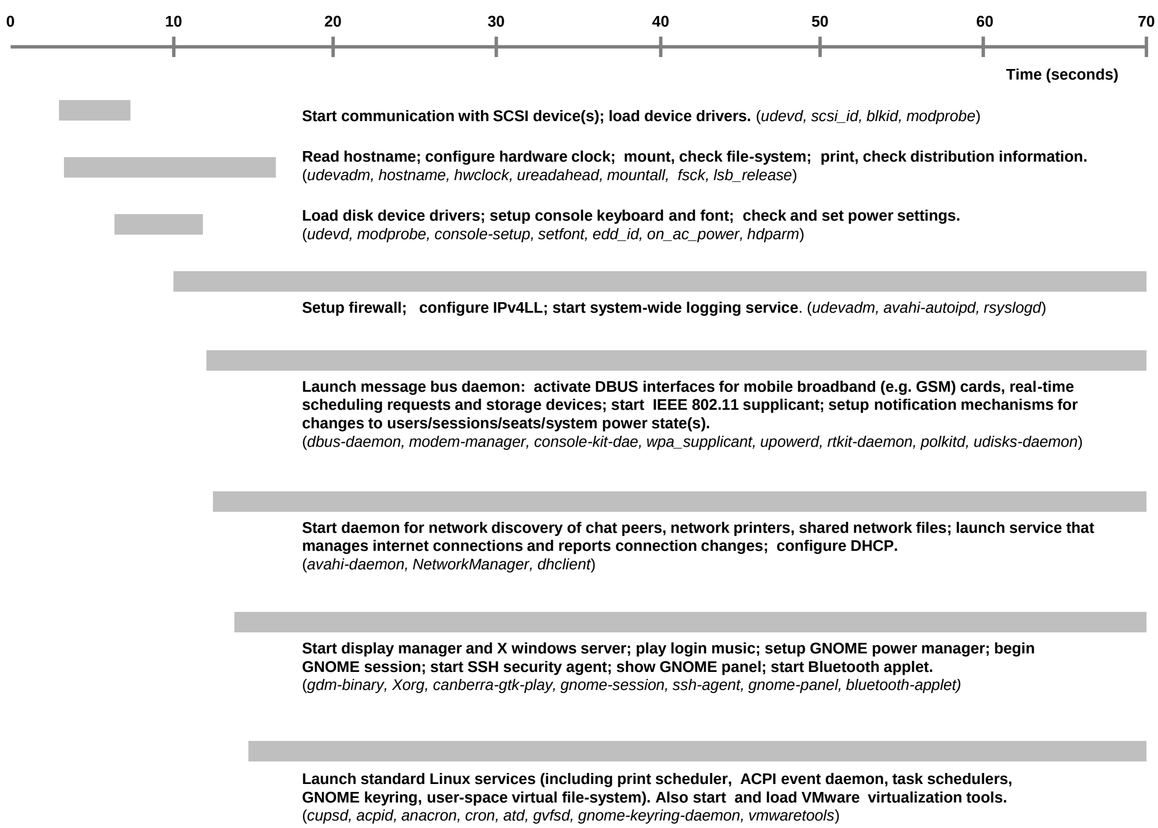

Figure 2-3: A summary of the actions performed by `init` for a booting VM; this figure has the same timeline (0-70 seconds) as Figures 2-1 and 2-2.

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

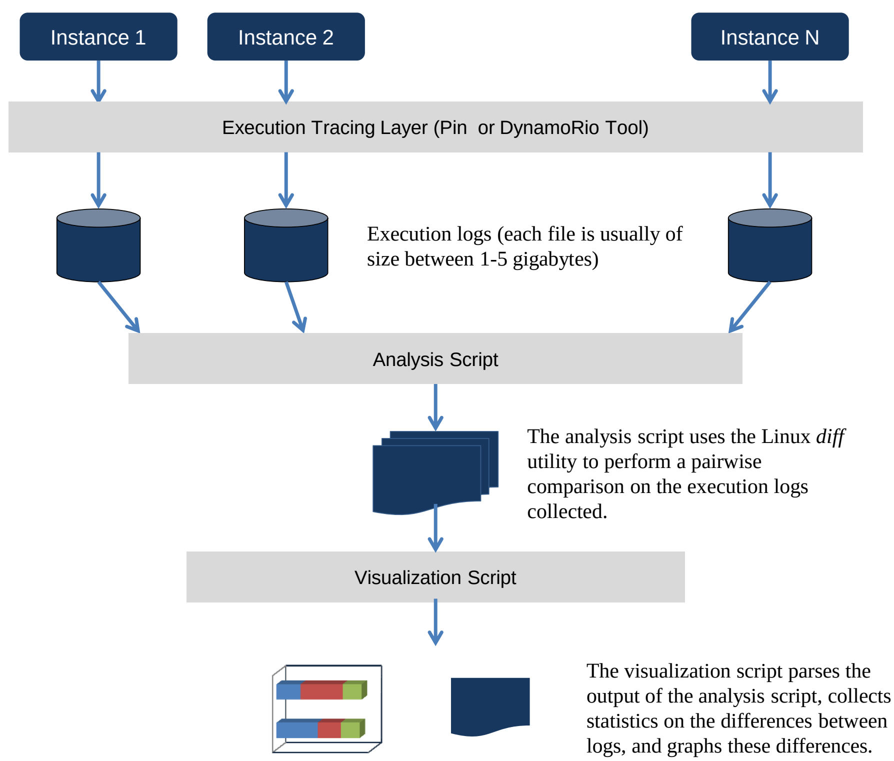

Figure 2-4: Steps involved in measuring execution nondeterminism.

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 together because this allowed us 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, which allows for 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 Analyzing a simple “Hello, world!” 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.

```
#include <stdio.h>

int
main(int argc, char* argv[])
{
  printf(‘‘Hello, world!\n’’);
  return 0;
}
```

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.

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 file generated by our Pin tool while running the "Hello, World" program. Our tool records and analyzes 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, dword ptr [esp]           $ ecx = 0xbffff2d4
_dl_make_stack_executable       0xb7ff6214  mov dword ptr [esp], eax
        Write *(UINT32*)0xbffff29c = 0xb6415b90
        Read 0xb7fff8f8 = *(UINT32*)0xbffff2a0
_dl_make_stack_executable       0xb7ff6217  mov eax, dword ptr [esp+0x4]       $ eax = 0xb7fff8f8
        Read 0xb6415b90 = *(UINT32*)0xbffff29c
_dl_make_stack_executable       0xb7ff621b  ret 0xc                            $ esp = 0xbffff2ac
__libc_start_main       0xb6415b90  push ebp                                   $ esp = 0xbffff2a8
        Write *(UINT32*)0xbffff2a8 = 0
__libc_start_main       0xb6415b91  mov ebp, esp                               $ ebp = 0xbffff2a8
__libc_start_main       0xb6415b93  push edi                                   $ esp = 0xbffff2a4
        Write *(UINT32*)0xbffff2a4 = 0x80482f0
__libc_start_main       0xb6415b94  push esi                                   $ esp = 0xbffff2a0
        Write *(UINT32*)0xbffff2a0 = 0x1
__libc_start_main       0xb6415b95  push ebx                                   $ esp = 0xbffff29c
        Write *(UINT32*)0xbffff29c = 0xb7ffeff4
__libc_start_main       0xb6415b96  call 0xb6415aaf                            $ esp = 0xbffff298
        Write *(UINT32*)0xbffff298 = 0xb6415b9b
        Read 0xb6415b9b = *(UINT32*)0xbffff298


 __NR_mmap2() called.
        addr = 0
        length = 4096
        prot = 3
        flags = 34
        fd = -1
        pgoffset = 0
        ret_val = b62dd000
 __NR_mmap2() returning.

 __NR_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] =
 __NR_write() returning.
```

Figure 2-6: An excerpt from the log files generated by the execution tracing layer. The top half shows the set of x86 instructions executed in user-space by the "Hello, world!" process, including instruction addresses, symbolic information (whenever available), affected register values and memory addresses. The lower half shows an excerpt from 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.

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 or least different, or have the median difference. In either case, these generated files can be used to detect and diagnose sources of nondeterminism in an application.

```
        Read 0xcb37 = *(UINT16*)0xbffff41b                                  |            Read 0xedf8 = *(UINT16*)0xbffff41b
.text   0xb7fe4dfe  movzx edx, word ptr [eax]          $ edx = 0xcb37       |    .text   0xb7fe4dfe  movzx edx, word ptr [eax]          $ edx = 0xedf8
        Write *(UINT16*)0xbffff10d = 0xcb37                                 |            Write *(UINT16*)0xbffff10d = 0xedf8
        Read 0x49 = *(UINT8*)0xbffff41d                                     |            Read 0x25 = *(UINT8*)0xbffff41d
.text   0xb7fe4e05  movzx eax, byte ptr [eax+0x2]      $ eax = 0x49         |    .text   0xb7fe4e05  movzx eax, byte ptr [eax+0x2]      $ eax = 0x25
        Write *(UINT8*)0xbffff10f = 0x49                                    |            Write *(UINT8*)0xbffff10f = 0x25
        Read 0x49cb3700 = *(UINT32*)0xbffff10c                              |            Read 0x25edf800 = *(UINT32*)0xbffff10c
.text   0xb7fe4e0c  mov esi, dword ptr [ebp-0x10]      $ esi = 0x49cb3700   |    .text   0xb7fe4e0c  mov esi, dword ptr [ebp-0x10]      $ esi = 0x25edf800
        Write *(UINT32*)0xb63ef8e4 = 0x49cb3700                             |            Write *(UINT32*)0xb63ef8e4 = 0x25edf800
        Read 0xc2a1e196 = *(UINT32*)0xbffff41f                              |            Read 0x6b875556 = *(UINT32*)0xbffff41f
.text   0xb7fe4e2a  mov esi, dword ptr [eax+0x4]       $ esi = 0xc2a1e196   |    .text   0xb7fe4e2a  mov esi, dword ptr [eax+0x4]       $ esi = 0x6b875556
        Write *(UINT32*)0xb63ef8e8 = 0xc2a1e196                             |            Write *(UINT32*)0xb63ef8e8 = 0x6b875556
        Write *(UINT32*)0xb7ffeef8 = 0xc2a1e196                             |            Write *(UINT32*)0xb7ffeef8 = 0x6b875556


.text   0xb7fe4ded  mov dword ptr [ebp-0x10], 0x0                                .text   0xb7fe4ded  mov dword ptr [ebp-0x10], 0x0
        Write *(UINT32*)0xbffff10c = 0                                                   Write *(UINT32*)0xbffff10c = 0
        Read 0xbffff41b = *(UINT32*)0xb7ffef24                                           Read 0xbffff41b = *(UINT32*)0xb7ffef24
.text   0xb7fe4df4  mov eax, dword ptr [ebx-0xd0]  $ eax = 0xbffff41b             .text   0xb7fe4df4  mov eax, dword ptr [ebx-0xd0]  $ eax = 0xbffff41b
.text   0xb7fe4dfa  test eax, eax  $ eflags = 0x286                              .text   0xb7fe4dfa  test eax, eax  $ eflags = 0x286
.text   0xb7fe4dfc  jz 0xb7fe4e51                                                .text   0xb7fe4dfc  jz 0xb7fe4e51
        Read 0xcb37 = *(UINT16*)0xbffff41b                                  |        Read 0xedf8 = *(UINT16*)0xbffff41b
.text   0xb7fe4dfe  movzx edx, word ptr [eax]  $ edx = 0xcb37               |    .text   0xb7fe4dfe  movzx edx, word ptr [eax]  $ edx = 0xedf8
.text   0xb7fe4e01  mov word ptr [ebp-0xf], dx                                   .text   0xb7fe4e01  mov word ptr [ebp-0xf], dx
        Write *(UINT16*)0xbffff10d = 0xcb37                                 |        Write *(UINT16*)0xbffff10d = 0xedf8
        Read 0x49 = *(UINT8*)0xbffff41d                                     |        Read 0x25 = *(UINT8*)0xbffff41d
.text   0xb7fe4e05  movzx eax, byte ptr [eax+0x2]  $ eax = 0x49             |    .text   0xb7fe4e05  movzx eax, byte ptr [eax+0x2]  $ eax = 0x25
```

Figure 2-7: Excerpts from the side-by-side diff files generated by the analysis script. The top half shows a few instructions at the start of the delta file; all these instructions are different in the two logs (as indicated by the | in the middle of the line). The bottom half shows the corresponding instructions in the union file. Conflicting instructions are marked with the color red in the union file (along with the | symbol); the other instructions are found in both logs.

Visualization Script

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

In particular, it derives three key metrics from 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 largest sequence of adjacent 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 instructions found 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 the two traces is arguably the most indicative 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. This is further discussed in chapter 5.

For the "Hello, world!" program, if ASLR is enabled, the two logs have very little overlap ($< 5\%$), and the common prefix and longest common substring are on the order of 10 instructions. With ASLR disabled, one may expect the two traces

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
Conflicts	0.02 percent
Conflicting Instructions	32

to look identical (because of the simplicity of the program), but there is still some nondeterminism in the instruction sequences (see Table 2.2 and Figure 2-8).

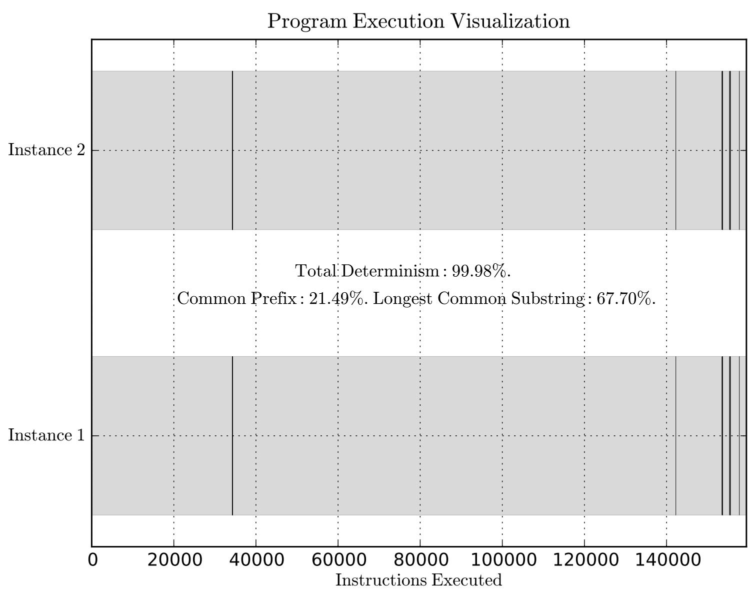

Figure 2-8: Visualization of "Hello, world!" program execution. The thin black lines represent conflicts between the two instances of the program.

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.

2.2.2 Alternative metrics for measuring 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 only considers instructions that originate nondeterminism or actively propagate it in computing the conflict ratio.

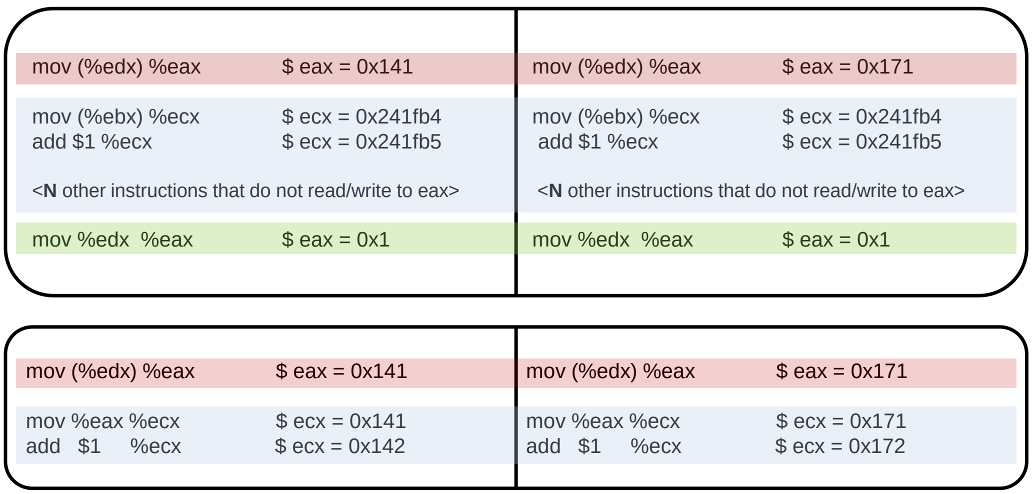

Figure 2-9: The top image shows an example of the cascade effect: the red instruction represents a real conflict in `eax`. The light-blue instructions have the same side-effects across the two logs because they don’t touch `eax`. The register state only converges after `eax` is written by the green instruction. 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 include the cascade effect, the measured conflict ratio in this trace excerpt is $(N+3)/(N+4)$ instead of $1/(N+4)$.

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.

As shown in Figure 2-9, the analysis script effectively simulates a taint analysis on register and memory contents to measure the true impact of any nondeterminism in a program. Grouping instructions that generate and then propagate nondeterminism makes it easier for us to diagnose the sources of nondeterminism.

2.3 Results for Linux services

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%

Table 2.2 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 in control flow after any initial conflict.

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.

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.

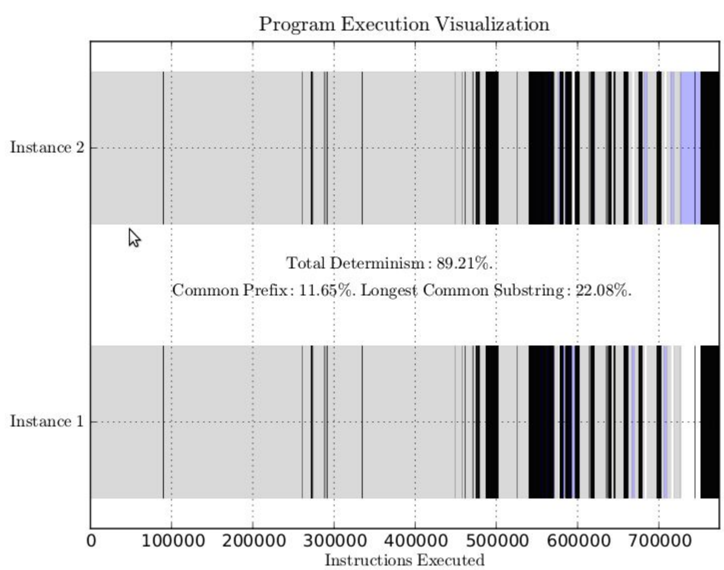

Figure 2-10: Visualization of `ntp` program execution. The thin black lines represent conflicts between the two instances of the program, whereas the thin blue lines represent control flow divergences.

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.

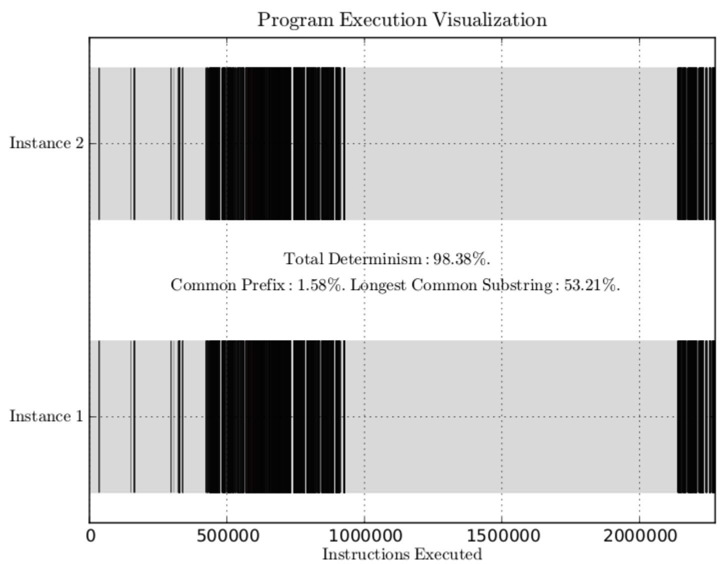

Figure 2-11: Visualization of `cron` program execution. The thin black lines represent conflicts between the two instances of the program.

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

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. $<< 1\%$) in our sample programs.

Table 2.3: Measuring burstiness of nondeterminism in Linux services

Application	Max. Consecutive Conflicts	Max. Control Flow Divergence
`ntp`, 14 loop iterations	0.03%	2%
`cron`, 30 loop iterations	0.08%	0.003%
`cups`, 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 first broadly summarizes the sources of nondeterminism in Linux services discovered through our experiments (Section 3.1). Next, it explains the dynamic instrumentation techniques we used to simulate deterministic execution in a boot storm scenario (Section 3.5). Finally, it discusses some of the limitations in our approach to deterministic execution (Section 3.6).

3.1 Sources of Nondeterminism

In this section, we describe the sources of nondeterminism that we 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, 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, the virtual addresses for the base executable, shared libraries, the heap, and the stack are different every time the program is run. 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 a simple "Hello, World!" program in *C* are almost entirely different when ASLR is enabled because of differences in 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 the 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 a random word 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 simply 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 transparent 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 via `longjmp`.

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 $s = p \oplus$ `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 $p = s \oplus$ `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.

This pointer guard has different values across multiple runs of a program, just like the `libc` 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 `XOR`ing `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 constitute a major source of nondeterminism in programs. Applications generally use pseudorandom number generators (PRNGs) for randomization and rely on the *seeds* to the PRNGs to be different across multiple program executions to generate truly random values. PRNG seeds are typically computed from several external 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 for the kernel random number generator.

- `AT_RANDOM` *bytes*: Making system calls to open, read and close the `'/dev/urandom'` file only for a few random bytes 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 below command-line arguments and environmental variables before a program starts executing.

- *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*: Many applications simply make a system call to get the current time or the current process ID, and use the returned value to seed their PRNGs.

- *Miscellaneous*: There are several creative ways to seed random number generators (e.g. read from *www.random.org*), but thankfully we have not observed them in the Linux services we have analyzed.

Thus, true nondeterminism in Linux services is caused from the external techniques used to seed PRNGs; if the seeds are different across multiple executions, PRNGs algorithms propagate this nondeterminsm much further.

3.1.3 Process Identification Layer

In the absence of a deterministic operating system layer, process IDs assigned to Linux services at boot-time are not predictable. For instance, a nondeterministic scheduler (Section 3.3) could lead to several possible process creation sequences and process ID assignments.

Given the unpredictability of process IDs, system calls that directly or indirectly interact with the process identification layer can cause divergences in 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) can return different values across 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 or writes to this address can cause conflicts.

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 3.2).

- *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 process IDs are used to seed PRNGs (Section 3.1.2), access the '/proc/[pid]' directory (Section 3.4), and create application-specific files or write to them. (Section 3.1.5).

3.1.4 Time

Concurrent runs of the same program will typically execute the same instructions at (slightly) different times. Clearly, any Linux system calls interact 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).

3.1.5 File I/O

If two executions of the same program read different file contents (e.g. cache files), then there will naturally be some execution divergence. However, 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. If those factors are controlled, file contents rarely differ.

Apart from minor differences in file contents, nondeterminism can arise from different 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 timestamps in the buffer written by the system call invariably conflict between 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 which file out of a group is the most recent, or whether a file has been updated and needs to be refreshed. Usually, simply replacing the time values with fixed ordinal values that preserve the ordering of timestamps is sufficient to control such nondeterminism.

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 `write`s 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 exections 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.

Ultimately, if two input files are semantically different between different executions of a program, then execution will inevitably diverge. However, the approaches mentioned in this chapter are typically sufficient to ensure that file contents rarely differ in Linux services, if at all.

3.1.6 Network I/O

The `libc` network initialization code loads several configuration files into memory (e.g. '`/etc/resolv.conf`'). The `dhclient` daemon updates '`/etc/resolv.conf`' with information from the DHCP server periodically in the background. Differences in the content, timestamps or lengths of such configuration files can cause nondeterminism. In fact, calls to `getaddrinfo` periodically `stat` configuration files (e.g. '`/etc/gai.conf`') to see if they have been changed in the background since they were last read. In our experiments, only the file timestamps of these configuration files vary between different executions, so the strategies described in Section 3.1.5 are sufficient to ensure deterministic execution.

IP addresses are resolved identically by concurrently executing programs in our experiments; however, if DNS-based load-balancing schemes are used, this may not be the case. In these rare cases, dynamic instrumentation can be used to enforce agreement between different executions.

If the contents read from sockets vary across different executions, dynamic instrumentation can be used to intercept Linux socket calls and modify their side-effects to be identical. If many concurrent executions are reading data from the same network source, this simply simulates the possibility that all instances see the same results as the first instance. While these semantics are sufficient for most Linux services, we have not needed to use them: in our experiments, we have only 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. `Netlink` can be used to pass networking information between kernel and user space. Netlink sockets use process IDs to identify communication endpoints, which can be different between executions. Furthermore, a netlink socket of the `NETLINK_ROUTE` family receives routing and link updates from the kernel. `libc` uses `RTM_NEWLINK` messages to discover the link interfaces in the computer. When a new interface is discovered or reported, the kernel also supplies interface statistics to `libc`, such as packets/bytes sent, dropped or received. These will obviously be different across different executions. However, we can easily use dynamic instrumentation to intercept `Netlink` communications and force these statistics to be identical.

A TCP/IPv4 connection consists of two endpoints, and each endpoint consists of an IP address and a port number. Therefore, when a client connects to a server, an established 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 clients uses its own IP and the server IP and server port are required. 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, the port number used is an ephemeral port number. Ephemeral ports are temporary ports assigned by the machine's IP stack, and are assigned from a dedicated range of ports for this purpose. When a connection terminates, an ephemeral port can be recycled. Since the underlying operating system is not deterministic, ephemeral port numbers used by Linux services tend be different across multiple runs. We mask this nondeterminism using dynamic instrumentation by changing the `bind` or `connect` system call arguments to explicitly request ports in the ephemeral range rather than letting the kernel assign them; alternatively, we can also virtualize ephemeral ports similar to how we virtualize process IDs.

3.1.7 I/O Polling Engines

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

- *pipes*, which 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 these pipes.
- the *listener socket*; if the program is a server, this is the socket that accepts incoming connections by calling `bind` and `listen`.
- open *client sockets*; if this program is a server, new requests from connected clients would arrive through such sockets.
- open *server sockets*; the program is a client, it would use this socket to send requests to the server.

The standard paradigm for implementing server programs is "one thread or process per client" because I/O operations are traditionally blocking. However, this approach scales poorly as the number of open file descriptors increases. Event-based I/O is increasingly used for simplicity by applications with many special file descriptors. In event-based I/O, one thread creates a set of file descriptors and specifies which events it cares about. It then waits for "readiness" notifications from the operating system for these file descriptors by using a system call such as `epoll`, `poll`, `select` or `kqueue`. Event-based I/O 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 extremely efficient because they simply report the set of file descriptors that are "ready" for I/O, preventing the need for the application to iterate through all the open file descriptors. For instance, a client socket would be ready for reading if a client made a new request by writing some data to its endpoint. Similarly, a server socket would be ready for writing if the outgoing buffer was flushed out or if the server had accepted the connection request.

Such event-based notification systems 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, it is not necessary that the same number of bytes are received by an application for each call to, say, `epoll`. Furthermore, when a timeout interval is specified by the application for polling fds, `select` many be completed or interrupted prematurely. In that case, `select` returns the remaining time interval, which can cause nondeterminism (Section 3.1.4).

To handle nondeterminism caused by the relative timing or the amount of data available in event-based polling engines, we can use dynamic instrumentation to intercept the corresponding system calls, and selectively deliver I/O events to order them identically across executions. In our experiments, this approach has been sufficient to achieve deterministic execution.

Other similar I/O mechanisms (e.g. AIO) would also create nondeterminism because of the variable absolute and relative timing of I/O events. Again, dynamic instrumentation techniques that reorder such events would be needed.

3.2 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), interrupts (e.g. from the shell), inter-process communication (e.g. parent sends `kill` to child process), or scheduled `alarm`s. Processes can register handlers for specific signals in order to respond to them.

Signals are clearly external to instructions executed by a single process, as such, they create nondeterminism in 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 in the execution of a program.

In order to overcome nondeterminism caused by signal delivery, we reorder signals across multiple executions using dynamic instrumentation. We also ensure that they are delivered at precisely the same instructions across different executions.

3.3 Concurrency

Multiple possible instruction interleavings of threads within a single program, or different processes in a single operating system are undoubtedly significant sources of nondeterminism.

Nondeterminism due to multi-threading has been extensively documented; there is a large 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 each thread individually. To overcome nondeterminism caused by multi-threading, we could incorporate deterministic logical clocks into our design.

Nondeterminism in the operating system scheduler is external to program execution, and manifests itself in different timing or ordering inter-process communications (e.g. through pipes, signals, or values written to file system logs). Using the schemes described in Sections 3.2 and 3.1.7. Existing work on deterministic operating systems can be extended to overcome these issues in a more systematic manner.

3.4 *Procfs*: The '/proc/' directory

User-space programs can access kernel data mostly from user-space via procfs, a hierarchical directory mounted at `'/proc/'`.

Because procfs is an interface to kernel data and system information that would otherwise be available via system calls (if at all), many of the sources of nondeterminsm already described can be propagated through it.

For instance, `'/proc/uptime'` contains the time statistics about system uptime; `'/proc/meminfo'` contains the statistics about kernel memory management; `'/proc/net/'` contains network statistics and IP addresses for interfaces; `'/proc/diskstats/'` contains statistcs about the attached disks.

Also, a process can access information about its open file descriptors through `/proc/[PID]/fdinfo` (e.g. cursor offset and status). `stat` on files in `'/proc/[PID]/'` can also reveal when a process was launched. Similarly, `/proc/[PID]/status` contains process-specific and highly variable statistics, e.g. number of involuntary context switches, memory usage, and parent ppid.

The solution is to use dynamic instrumentation to intercept and modify reads from procfs.

3.5 Simulating Determinism

Existing record-and-replay systems sometimes overcome ASLR by forcing the operating system to use the same address space layout across different runs. A slightly more complicated approach would involve using base/offset computations to translate equivalent addresses between two different executions. We simply disabled ASLR for our experiments using the following command: `sudo kernel.randomize_va_space=0`.

Dynamic instrumentation can be used to force canary values to agree across distinct executions of the same program: instructions that initialize `gs:0x14` can be modified or replaced, or the `AT_RANDOM` bytes provided by the kernel can be modified before they are read by the application, or reads from `'/dev/urandom'` can be intercepted.

In any case, the value of the pointer guard can be made to agree across different instances of the same program via dynamic instrumentation: the instructions that initialize `gs:0x18` can be modified or replaced, or the `rdtsc` instruction can be intercepted and emulated, or the `AT_RANDOM` bytes provided by the kernel can be replaced before they are ever read.

To overcome nondeterminism resulting from randomization, we need to intercept the standard techniques used by programs to seed PRNGs. Dynamic instrumentation can be used to intercept reads from `'/dev/urandom'` and emulate `rdtsc` instructions. Nondeterminism from process IDs can be controlled by virtualizing the process ID layer (Section 3.1.3), and nondeterminism from time can be controlled by intercepting time-related system calls and forcing agreement between concurrently executing instances (Section 3.1.4).

Figure 3-1: We intercept all system calls and communications between the Linux user and kernel space; we translate between real and virtual process IDs to trick the kernel and the user-space programs.

Figure 3-1 shows how we can use dynamic instrumentation techniques to virtualize and determinize the process ID layer in Linux. Using these techniques, we were able to avoid modifying the Linux operating system and existing programs altogether.

To overcome nondeterminism from time-related system calls, we can use dynamic instrumentation to force agreement between any timestamps returned across concurrent executions (taking care to preserve monotonicity). This creates the illusion that multiple executions are occuring precisely at the same time. When timestamps returned from system calls are only compared, they can be replaced with deterministic ordinal values that perserve the comparison (e.g 0 or 1).

3.6 Limitations of Deterministic Execution

3.7 Summary

Chapter 4

Case Studies of Linux Services

This chapter summarizes the context and extent of non-determinism found in three Linux services (`cron, cupsd` and `ntp`) in detail.

4.1 Cups

4.2 Cron

4.3 Ntp

4.4 Summary

THIS PAGE INTENTIONALLY LEFT BLANK

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. Guttag, 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.