beyond breakpoints

A Tour of Dynamic Analysis

Nathan Taylor, Fastly
http://nathan.dijkstracula.net
@dijkstracula

fastly®

Xen™

T. Ball: **The Concept of Dynamic Analysis** ESEC '99

The Concept of Dynamic Analysis

Thomas Ball

Bell Laboratories
Lucent Technologies
tball@research.bell-labs.com

Abstract. Dynamic analysis is the analysis of the properties of a running program. In this paper, we explore two new dynamic analyses based on program profiling:

- *Frequency Spectrum Analysis.* We show how analyzing the frequencies of program entities in a single execution can help programmers to decompose a program, identify related computations, and find computations related to specific input and output characteristics of a program.
- *Coverage Concept Analysis.* Concept analysis of test coverage data computes dynamic analogs to static control flow relationships such as domination, postdomination, and regions. Comparison of these dynamically computed relationships to their static counterparts can point to areas of code requiring more testing and can aid programmers in understanding how a program and its test sets relate to one another.

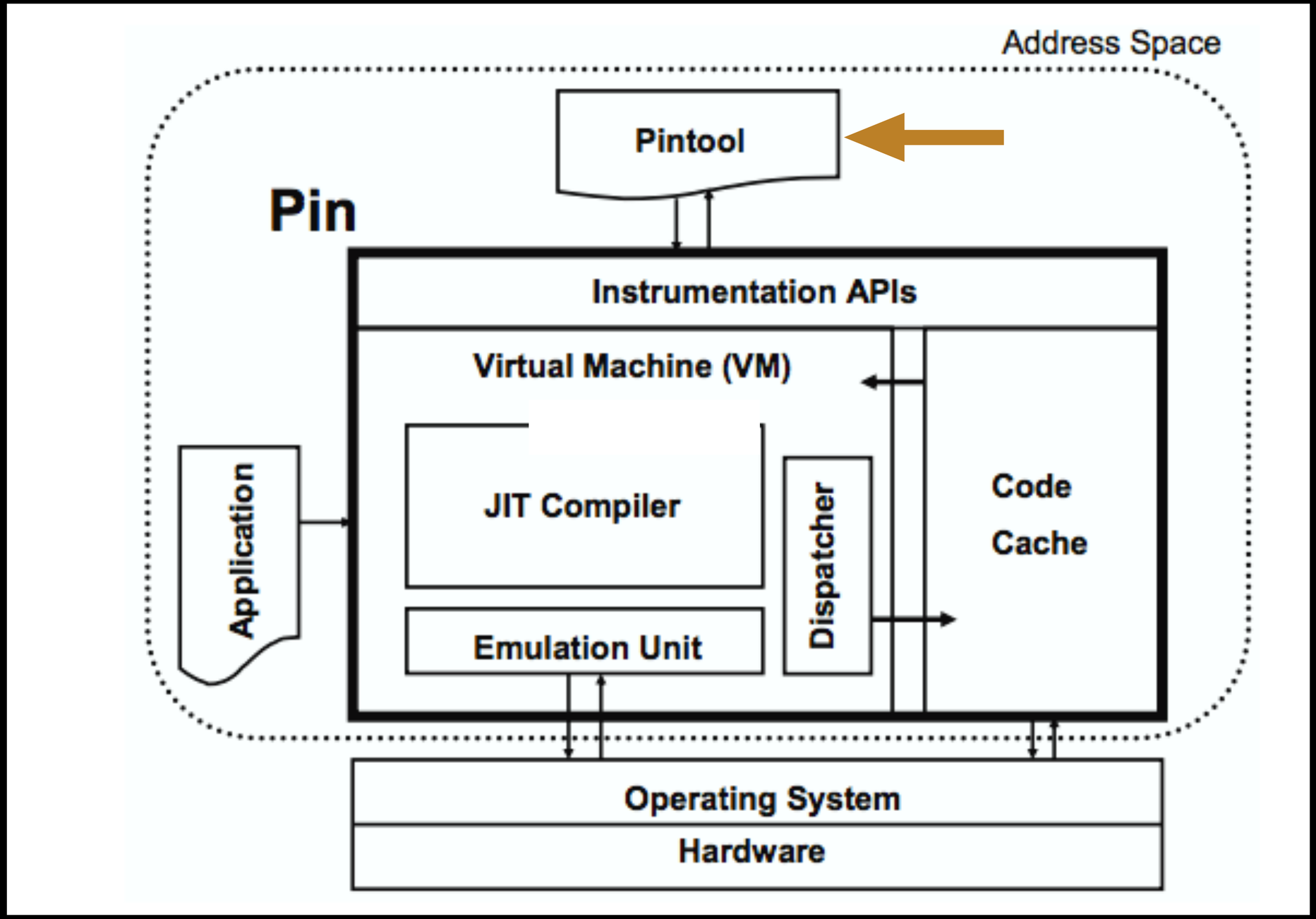
1 Introduction

Dynamic analysis is the analysis of the properties of a running program. In contrast to static analysis, which examines a program's text to derive properties that hold for all executions, dynamic analysis derives properties that hold for one or more executions by examination of the running program (usually through program instrumentation [14]). While dynamic analysis cannot prove that a program satisfies a particular property, it can detect violations of properties as well as provide useful information to programmers about the behavior of their programs, as this paper will show.

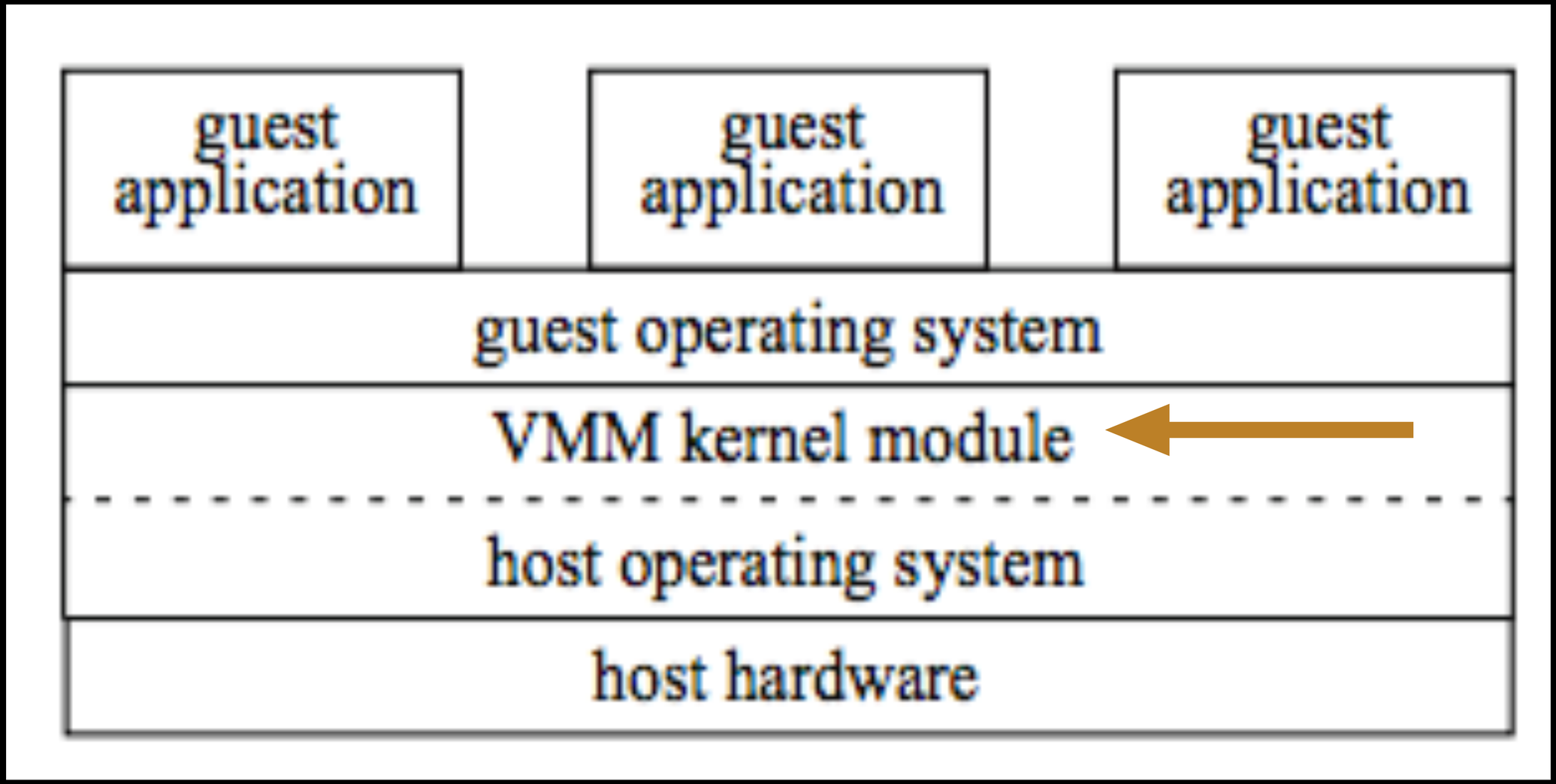
The usefulness of dynamic analysis derives from two of its essential characteristics:

- *Precision of information*: dynamic analysis typically involves instrumenting a program to examine or record certain aspects of its run-time state. This instrumentation can be tuned to collect precisely the information needed

"Dynamic analysis is the analysis of the properties of a running program [...] (usually through program instrumentation)."



From Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation



From ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay

classic use-cases

- Debugging a program crash or race condition
- Input to understand a new codebase
- Analysis of obfuscated software or malware

who cares

- Huge gulf between what tools practitioners use in industry, and academic dynamic analysis work
- Many techniques shared with other cool areas like OS virtualization and compiler theory
- Techniques have different performance tradeoffs; knowing the lay of the land will help you choose the right tool for your problem

dynamic vs static analysis

- Static analysis is a conservative approximation of runtime behavior
  - Programs are only partially known (dynamic linking; user input)
  - Imprecise: consider a program that typechecks but still has a bug
  - Language specifications often assume a single thread of execution (or, don’t specify at all): “valid” programs can still allow race conditions!

can’t prove correctness

but can demonstrate failure

today’s topics

- Omniscient Debugging and State Tracking
- Analyzing Concurrent Systems
- Areas of Future Work

for each topic...

- What open-source tooling exists?
- How does it work under the hood?
- What contributions has academia made?

```
➜  /tmp gcc foo.c
```

```
➜  /tmp gcc foo.c
➜  /tmp ./a.out
[1]    4233 segmentation fault  ./a.out
```

```
➜  /tmp gcc foo.c
➜  /tmp ./a.out
[1]    4233 segmentation fault  ./a.out
➜  /tmp gcc -g foo.c
➜  /tmp gdb ./a.out
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
(gdb) run
```

```
➜  /tmp gcc foo.c
➜  /tmp ./a.out
[1]    4233 segmentation fault  ./a.out
➜  /tmp gcc -g foo.c
➜  /tmp gdb ./a.out
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
(gdb) run
Starting program: /tmp/a.out

Program received signal SIGSEGV, Segmentation fault.
0x00000000004004fd in baz () at foo.c:5
5           *foo = 42;
```

```
Starting program: /tmp/a.out

Program received signal SIGSEGV, Segmentation fault.
0x00000000004004fd in baz () at foo.c:5
5               *foo = 42;
(gdb) bt
#0  0x00000000004004fd in baz () at foo.c:5
#1  0x0000000000400513 in bar () at foo.c:9
#2  0x0000000000400523 in foo () at foo.c:13
#3  0x000000000040053e in main (argc=1, argv=0x7fffffffe438) at foo.c:17
(gdb) inf loc
foo = 0x7
(gdb)
```

J. Mickens: **The Night Watch** ;login Nov 2013

The Night Watch

JAMES MICKENS

James Mickens is a researcher in the Distributed Systems group at Microsoft's Redmond lab. His current research focuses on web applications, with an emphasis on the design of JavaScript frameworks that allow developers to diagnose and fix bugs in widely deployed web applications. James also works on fast, scalable storage systems for datacenters. James received his PhD in computer science from the University of Michigan, and a bachelor's degree in computer science from Georgia Tech. mickens@microsoft.com

As a highly trained academic researcher, I spend a lot of time trying to advance the frontiers of human knowledge. However, as someone who was born in the South, I secretly believe that true progress is a fantasy, and that I need to prepare for the end times, and for the chickens coming home to roost, and fast zombies, and slow zombies, and the polite zombies who say "sir" and "ma'am" but then try to eat your brain to acquire your skills. When the revolution comes, I need to be prepared; thus, in the quiet moments, when I'm not producing incredible scientific breakthroughs, I think about what I'll do when the weather forecast inevitably becomes RIVERS OF BLOOD ALL DAY EVERY DAY. The main thing that I ponder is who will be in my gang, because the likelihood of post-apocalyptic survival is directly related to the size and quality of your rag-tag group of associates. There are some obvious people who I'll need to recruit: a locksmith (to open doors); a demolitions expert (for when the locksmith has run out of ideas); and a person who can procure, train, and then throw snakes at my enemies (because, in a world without hope, snake throwing is a reasonable way to resolve disputes). All of these people will play a role in my ultimate success as a dystopian warlord philosopher. However, the most important person in my gang will be a systems programmer. A person who can debug a device driver or a distributed system is a person who can be trusted in a Hobbesian nightmare of breathtaking scope; a systems programmer has seen the terrors of the world and understood the intrinsic horror of existence. The systems programmer has written drivers for buggy devices whose firmware was implemented by a drunken child or a sober goldfish. The systems programmer has traced a network problem across eight machines, three time zones, and a brief diversion into Amish country, where the problem was transmitted in the front left hoof of a mule named Deliverance. The systems programmer has read the kernel source, to better understand the deep ways of the universe, and the systems programmer has seen the comment in the scheduler that says "DOES THIS WORK LOL," and the systems programmer has wept instead of LOLed, and the systems programmer has submitted a kernel patch to restore balance to The Force and fix the priority inversion that was causing MySQL to hang. A systems programmer will know what to do when society breaks down, because the systems programmer already lives in a world without law.

Listen: I'm not saying that other kinds of computer people are useless. I believe (but cannot prove) that PHP developers have souls. I think it's great that database people keep trying to improve select-from-where, even though the only queries that cannot be expressed using select-from-where are inappropriate limericks from "The Canterbury Tales." In some way that I don't yet understand, I'm glad that theorists are investigating the equivalence between five-dimensional Turing machines and Edward Scissorhands. In most situations, GUI designers should not be forced to fight each other with tridents and nets as I yell "THERE ARE NO MODAL DIALOGS IN SPARTA." I am like the Statue of Liberty: I accept everyone, even the wretched and the huddled and people who enjoy Haskell. But when things get tough, I need mission-critical people; I need a person who can wear night-vision goggles and descend from a helicopter on ropes and do classified things to protect my freedom while country music plays in the background. A systems person can do that. I can realistically give a kernel hacker a nickname like "Diamondback" or "Zeus Hammer." In contrast, no one has ever said, "These semi-transparent icons are really semi-transparent! IS THIS THE using chewing tobacco. As a systems hacker, you must be prepared to do savage things, unspeakable things, to kill runaway threads with your bare hands, to write directly to network ports using telnet and an old copy of an RFC that you found in the Vatican. When you debug systems code, there are no high-level debates about font choices and the best kind of turquoise, because this is the Old Testament, an angry and monochromatic world, and it doesn't matter whether your Arial is Bold or Condensed when people are covered in boils and pestilence and Egyptian pharaoh oppression. HCI people discover bugs by receiving a concerned email from their therapist. Systems people discover bugs by waking up and discovering that their first-born children are missing and "ETIMEDOUT" has been written in blood on the wall.

What is despair? I have known it—hear my song. Despair is when you're debugging a kernel driver and you look at a memory dump and you see that a pointer has a value of 7. THERE IS NO HARDWARE ARCHITECTURE THAT IS ALIGNED ON 7. Furthermore, 7 IS TOO SMALL AND ONLY EVIL CODE WOULD TRY TO ACCESS SMALL NUMBER MEMORY. Misaligned, small-number memory accesses have stolen

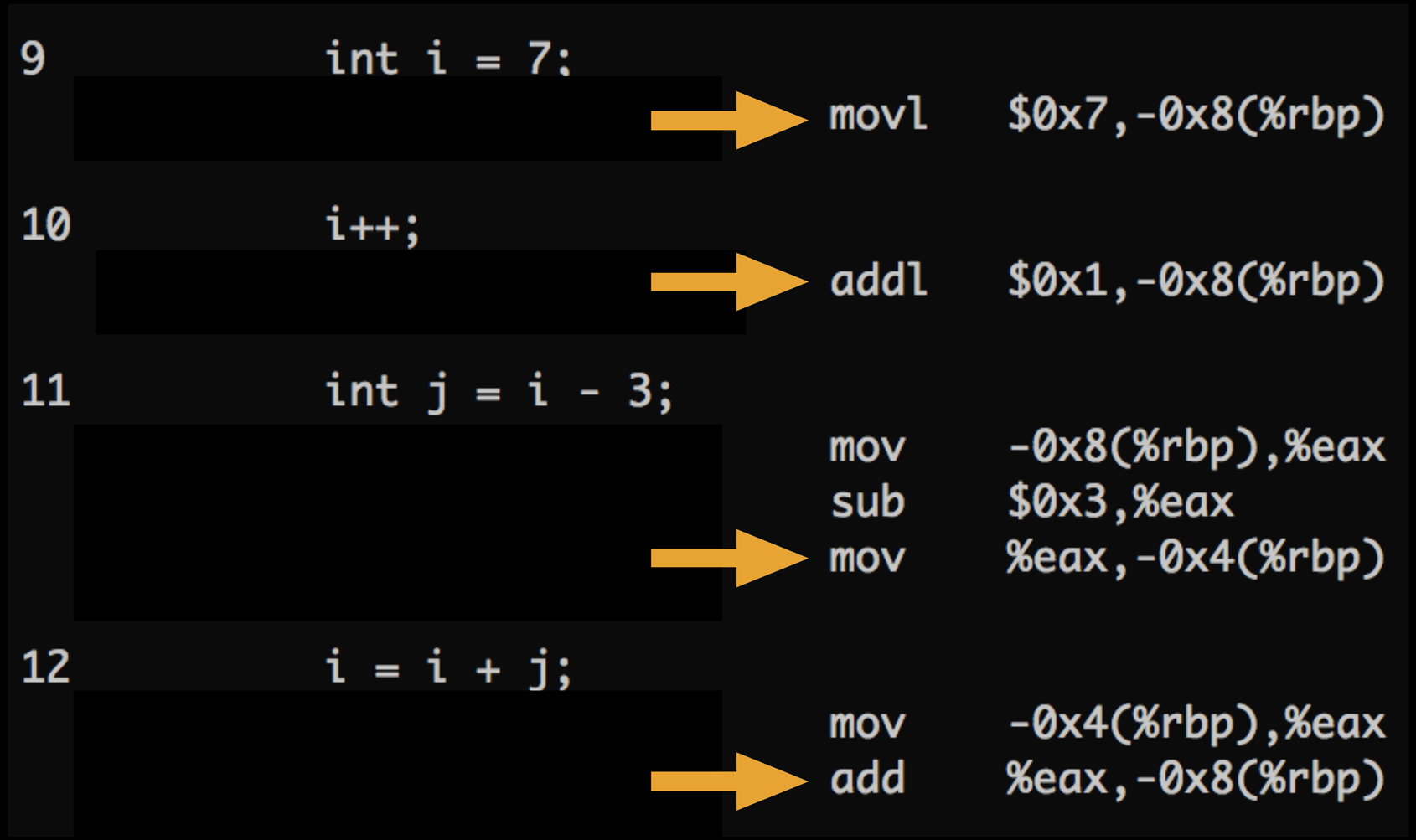
> "Despair is when you're debugging a kernel driver and you look at a memory dump and you see that a pointer has a value of 7. THERE IS NO HARDWARE ARCHITECTURE THAT IS ALIGNED ON 7. Furthermore, 7 IS TOO SMALL AND ONLY EVIL CODE WOULD TRY TO ACCESS SMALL NUMBER MEMORY."

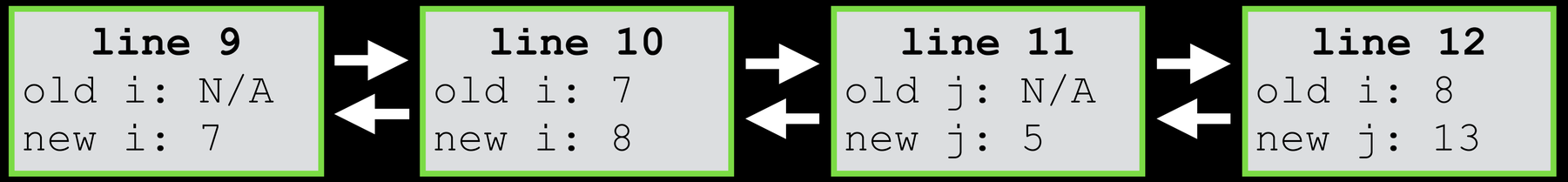
```
(gdb) run
Starting program: /tmp/a.out

Program received signal SIGSEGV, Segmentation fault.
0x00000000004004fd in baz () at foo.c:5
5               *foo = 42;
(gdb) bt
#0  0x00000000004004fd in baz () at foo.c:5
#1  0x0000000000400513 in bar () at foo.c:9
#2  0x0000000000400523 in foo () at foo.c:13
#3  0x000000000040053e in main (argc=1, argv=0x7fffffffe438)
(gdb) inf loc
foo = 0x7
(gdb)
```

Time-Traveling Debugging

```
 8 void bar() {
 9     int i = 7;
10     i++;
11     int j = i - 3;
12     i = i + j;
```






```
9           int i = 7;
                                movl    $0x7,-0x8(%rbp)

10          i++;
                                addl    $0x1,-0x8(%rbp)

11          int j = i - 3;
                                mov     -0x8(%rbp),%eax
                                sub     $0x3,%eax
                                mov     %eax,-0x4(%rbp)

12          i = i + j;
                                mov     -0x4(%rbp),%eax
                                add     %eax,-0x8(%rbp)
```

```
68 /* These are the core structs of the process record functionality.
69
70    A record_full_entry is a record of the value change of a register
71    ("record_full_reg") or a part of memory ("record_full_mem").  And each
72    instruction must have a struct record_full_entry ("record_full_end")
73    that indicates that this is the last struct record_full_entry of this
74    instruction.
75
76    Each struct record_full_entry is linked to "record_full_list" by "prev"
77    and "next" pointers.  */
141 struct record_full_entry
142 {
143   struct record_full_entry *prev;
144   struct record_full_entry *next;
145   enum record_full_type type;
146   union
147   {
148     /* reg */
149     struct record_full_reg_entry reg;
150     /* mem */
151     struct record_full_mem_entry mem;
152     /* end */
153     struct record_full_end_entry end;
154   } u;
155 };
```

gdb/record-full.c

```
➜ /tmp gdb ./a.out
```

Hofer et al. **Design And Implementation of a Backward-In-Time Debugger**

NODe '06

Design and Implementation of a Backward-In-Time Debugger*

Christoph Hofer, Marcus Denker
Software Composition Group
University of Bern, Switzerland
www.iam.unibe.ch/~scg

Stéphane Ducasse
LISTIC
Université de Savoie, France
www.listic.univ-savoie.fr

Abstract:
Traditional debugging and stepping execution trace are well-accepted techniques to understand deep internals about a program. However in many cases navigating the stack trace is not enough to find bugs, since the cause of a bug is often not in the stack trace anymore and old state is lost, so out of reach from the debugger. In this paper, we present the design and implementation of a backward-in-time debugger for a dynamic language, *i.e.,* a debugger that allows one to navigate back the history of the application. We present the design and implementation of a backward-in-time debugger called UNSTUCK and show our solution to key implementation challenges.

1 Introduction

Debuggers offer the ability to stop a program at a chosen place, either due to an error or an explicit request (breakpoint). They provide the current states of the involved objects together with a stack trace. However, while stepping through the code is a powerful technique to get a deep understanding of a certain functionality [DDN02], in many cases this information is not enough to find bugs. The programmer is often forced to build new hypotheses about the possible cause of the bugs, set new breakpoints and restart the program to find the source of the problem. Often several iterations are necessary and it may be difficult to recreate the exact same context [LHS99].

The questions a programmer has are often: *"where was this variable set?"*, *"why is this object reference nil?"* or *"what was the previous state of that object?"*. A static debugger cannot answer these questions, since it has only access to the current execution stack. There is no possibility to backtrack the state of an object or to find out why especially this object was passed to a method. The Omniscient Debugger is a first attempt to answer these problems [Lew03], however it is limited to java and instrumentation is done at bytecode load time.

To understand the challenges faced by building a backward in time debugger, *i.e.,* a debugger that allows one to query the state history of a program, we developed a backward

README.md

librip

Librip is a minimal-overhead API for instruction-level tracing in highly concurrent software. It is released under the Apache 2.0 license.

Impetus

Software with many thousands of threads and / or coroutines suffers from lack of run-time and post-mortem visibility with debugging tools. When many thousands of threads are present in a software system, it may be impossible to attach a debugger to an apparently stuck process. With many thousands of coroutines, even with a debugger, it can be difficult to even find a stuck coroutine on myriad scheduling lists -- especially if it is erroneously waiting for data in the kernel!

This library attempts to solve these problems through an API that is reasonably efficient in terms of both CPU and memory requirements -- enough so that traces can be take in production.

Design

As the name suggests, this library provides an interface for snapshots of the instruction pointer. These snapshots are stored in a per-thread ring buffer, and contain a packed counter / function address. (Additional interfaces allow registration for coroutines, but require additional runtime support.)

Currently, only Linux on amd64 architectures is supported. Patches for other platforms, operating systems, and compilers are more than welcome.

Efficient time-traveling

King et al. **Debugging Operating Systems with Time-Traveling Virtual Machines**

Usenix ‘05

Debugging operating systems with time-traveling virtual machines

Samuel T. King, George W. Dunlap, and Peter M. Chen
University of Michigan

Abstract

Operating systems are difficult to debug with traditional cyclic debugging. They are non-deterministic; they run for long periods of time; they interact directly with hardware devices; and their state is easily perturbed by the act of debugging. This paper describes a time-traveling virtual machine that overcomes many of the difficulties associated with debugging operating systems. Time travel enables a programmer to navigate backward and forward arbitrarily through the execution history of a particular run and to replay arbitrary segments of the past execution. We integrate time travel into a general-purpose debugger to enable a programmer to debug an OS in reverse, implementing commands such as reverse breakpoint, reverse watchpoint, and reverse single step. The space and time overheads needed to support time travel are reasonable for debugging, and movements in time are fast enough to support interactive debugging. We demonstrate the value of our time-traveling virtual machine by using it to understand and fix several OS bugs that are difficult to find with standard debugging tools. Reverse debugging is especially helpful in finding bugs that are fragile due to non-determinism, bugs in device drivers, bugs that require long runs to trigger, bugs that corrupt the stack, and bugs that are detected after the relevant stack frame is popped.

1 Introduction

Computer programmers are all-too-familiar with the task of debugging complex software through a combination of detective work, guesswork, and systematic search. Tracking down a bug generally starts with running a program until an error in the program manifests as a fault. The programmer[1] then seeks to start from the fault (the manifestation of the error) and work backward to the cause of the fault (the programming error itself). Cyclic debugging is the classic way to work backward toward the error. In cyclic debugging, a programmer uses a debugger or output statements to examine the state of the program at a given point in its execution. Armed with this information, the programmer then re-runs the program, stops it at an earlier point in its execution history, examines the state at this point, then iterates.

Unfortunately, this classic approach to debugging is difficult to apply when debugging operating systems. Many aspects of operating systems make them difficult to debug: they are non-deterministic; they run for long periods of time; the act of debugging may perturb their state; and they interact directly with hardware devices.

First, operating systems are non-deterministic. Their execution is affected by non-deterministic events such as the interleaving of multiple threads, interrupts, user input, network input, and the perturbations of state caused by the programmer who is debugging the system. This non-determinism makes cyclic debugging infeasible because the programmer cannot re-run the system to examine the state at an earlier point.

Second, operating systems run for long periods of time, such as weeks, months, or even years. Re-running the system in cyclic debugging would thus be infeasible even if the OS were completely deterministic.

Third, the act of debugging may perturb the state of the operating system. The converse is also true: a misbehaving operating system may corrupt the state of the debugger. These interactions are possible because the operating system is traditionally the lowest level of software on a computer, so the debugger's code and data is not isolated from the OS (unless the debugger uses specialized hardware such as an in-circuit emulator). Even remote kernel debuggers depend on some basic functionality in the debugged OS, such as reading and writing memory locations, setting and handling breakpoints, and communicating with the remote debugger (e.g., through the serial line). Using this basic functionality may be impossible on a sick OS. A debugger also needs assistance from the OS to access hardware devices, and this functionality may not work on a sick OS.

Finally, operating systems interact directly with hardware devices. Devices are sources of non-determinism that hinder cyclic debugging; they return data and generate interrupts that may change between runs. Devices may also fail due to timing dependencies if a programmer pauses during a debugging session.

In this paper, we describe how to use *time-traveling virtual machines* to overcome many of the difficulties as-

[1] In this paper, "programmer" refers to the person debugging the system, and "debugger" refers to the programming tool (e.g., `gdb`) used by the programmer to examine and control the program.

> “...operating systems run for long periods of time, such as weeks, months, or even years.”

King et al. **Debugging Operating Systems with Time-Traveling Virtual Machines**

Usenix ‘05

Debugging operating systems with time-traveling virtual machines

Samuel T. King, George W. Dunlap, and Peter M. Chen
University of Michigan

Abstract

Operating systems are difficult to debug with traditional cyclic debugging. They are non-deterministic; they run for long periods of time; they interact directly with hardware devices; and their state is easily perturbed by the act of debugging. This paper describes a time-traveling virtual machine that overcomes many of the difficulties associated with debugging operating systems. Time travel enables a programmer to navigate backward and forward arbitrarily through the execution history of a particular run and to replay arbitrary segments of the past execution. We integrate time travel into a general-purpose debugger to enable a programmer to debug an OS in reverse, implementing commands such as reverse breakpoint, reverse watchpoint, and reverse single step. The space and time overheads needed to support time travel are reasonable for debugging, and movements in time are fast enough to support interactive debugging. We demonstrate the value of our time-traveling virtual machine by using it to understand and fix several OS bugs that are difficult to find with standard debugging tools. Reverse debugging is especially helpful in finding bugs that are fragile due to non-determinism, bugs in device drivers, bugs that require long runs to trigger, bugs that corrupt the stack, and bugs that are detected after the relevant stack frame is popped.

1 Introduction

Computer programmers are all-too-familiar with the task of debugging complex software through a combination of detective work, guesswork, and systematic search. Tracking down a bug generally starts with running a program until an error in the program manifests as a fault. The programmer[1] then seeks to start from the fault (the manifestation of the error) and work backward to the cause of the fault (the programming error itself). Cyclic debugging is the classic way to work backward toward the error. In cyclic debugging, a programmer uses a debugger or output statements to examine the state of the program at a given point in its execution. Armed with this information, the programmer then re-runs the program, stops it at an earlier point in its execution history, examines the state at this point, then iterates.

Unfortunately, this classic approach to debugging is difficult to apply when debugging operating systems. Many aspects of operating systems make them difficult to debug: they are non-deterministic; they run for long periods of time; the act of debugging may perturb their state; and they interact directly with hardware devices.

First, operating systems are non-deterministic. Their execution is affected by non-deterministic events such as the interleaving of multiple threads, interrupts, user input, network input, and the perturbations of state caused by the programmer who is debugging the system. This non-determinism makes cyclic debugging infeasible because the programmer cannot re-run the system to examine the state at an earlier point.

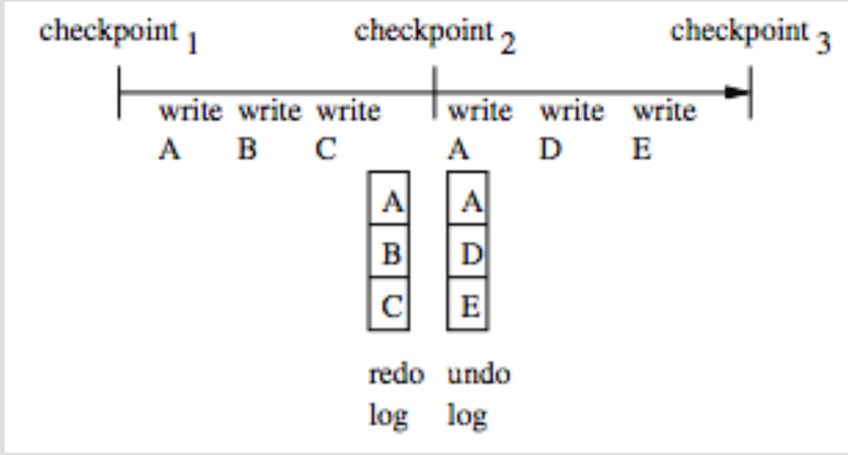
Second, operating systems run for long periods of time, such as weeks, months, or even years. Re-running the system in cyclic debugging would thus be infeasible even if the OS were completely deterministic.

Third, the act of debugging may perturb the state of the operating system. The converse is also true: a misbehaving operating system may corrupt the state of the debugger. These interactions are possible because the operating system is traditionally the lowest level of software on a computer, so the debugger's code and data is not isolated from the OS (unless the debugger uses specialized hardware such as an in-circuit emulator). Even remote kernel debuggers depend on some basic functionality in the debugged OS, such as reading and writing memory locations, setting and handling breakpoints, and communicating with the remote debugger (e.g., through the serial line). Using this basic functionality may be impossible on a sick OS. A debugger also needs assistance from the OS to access hardware devices, and this functionality may not work on a sick OS.

Finally, operating systems interact directly with hardware devices. Devices are sources of non-determinism that hinder cyclic debugging; they return data and generate interrupts that may change between runs. Devices may also fail due to timing dependencies if a programmer pauses during a debugging session.

In this paper, we describe how to use *time-traveling virtual machines* to overcome many of the difficulties as-

[1] In this paper, "programmer" refers to the person debugging the system, and "debugger" refers to the programming tool (e.g., `gdb`) used by the programmer to examine and control the program.



King et al. **Debugging Operating Systems with Time-Traveling Virtual Machines**

Usenix '05

3 Time-traveling virtual machines

A time-traveling virtual machine should have two capabilities. First, it should be able to reconstruct the complete state of the virtual machine at any point in a run, where a run is defined as the time from when the virtual machine was powered on to the last instruction it executed. Second, it should be able to start from any point in a run and from that point replay the same instruction stream that was executed during the original run from that point. This section describes how TTVM achieves these capabilities through a combination of logging, replay, and checkpointing.

3.1 Logging and replaying a VM

The foundational capability in TTVM is the ability to replay a run from a given point in a way that matches the original run instruction for instruction. Replay causes the virtual machine to transition through the same states as it went through during the original run; hence replay enables one to reconstruct the complete state of the virtual machine at any point in the run. TTVM uses the ReVirt logging/replay system to provide this capability [9]. This section briefly summarizes how ReVirt logs and replays the execution of a virtual machine.

A virtual machine can be replayed by starting from a checkpoint, then replaying all sources of nondeterminism [5, 9]. For UML, the sources of nondeterminism are external input from the network, keyboard, and real-time clock and the timing of virtual interrupts. The VMM replays network and keyboard input by logging the calls that read these devices during the original run and regenerating the same data during the replay run. Likewise, we configure the CPU to cause reads of the real-time clock to trap to the VMM, where they can be logged or regenerated.

To replay a virtual interrupt, ReVirt logs the instruction in the run at which it was delivered and re-delivers the interrupt at this instruction during replay. This point is identified uniquely by the number of branches since the start of the run and the address of the interrupted instruction [19]. ReVirt uses a performance counter on the Intel Pentium 4 CPU to count the number of branches during logging, and it uses the same performance counter and instruction breakpoints to stop at the interrupted instruction during replay. Replaying interrupts enables ReVirt to replay the scheduling order of multi-threaded guest operating systems and applications, as long as the VMM exports the abstraction of a uniprocessor virtual machine [22]. Researchers are investigating ways to support replay on multiprocessors [29].

3.2 Host device drivers in the guest OS

In general, VMMs export a limited set of virtual devices. Some VMMs export virtual devices that exist in hardware (e.g., VMware Workstation exports an emulated AMD Lance Ethernet card); others (like UML) export virtual devices that have no hardware equivalent. Exporting a limited set of virtual devices to the guest OS is usually considered a benefit of virtual-machine systems, because it frees guest OSs from needing device drivers for myriad host devices [26]. However, when using virtual machines to debug operating systems, the limited set of virtual devices prevents programmers from using and debugging drivers for real devices; programmers can only debug the architecture-independent portion of the guest OS. There are two ways to address this limitation and enable the programmer to run and debug real device drivers in a guest OS. With both strategies, real device drivers can be included in the guest OS without being modified or re-compiled.

The first way to run a real device driver in the guest OS is for the VMM to provide a software emulator for that device. The device driver issues the normal set of I/O instructions: `IN`/`OUT` instructions, memory-mapped I/O, DMA commands, and interrupts. The VMM traps these privileged instructions and forwards them to/from the software device emulator. With this strategy, ReVirt can log and replay device driver code in the same way it logs and replays the rest of the guest OS. If one runs the VMM's software device emulator above ReVirt's logging system (and above the checkpoint system described in Section 3.3), ReVirt will guide the emulator and device driver code through the same instruction sequence during replay as they executed during logging. While this first strategy fits in well with the existing ReVirt system, it only works if one has an accurate software emulator for the device whose driver one wishes to debug.

We modified UML to provide a second way to run real device drivers in the guest OS, which works even when no software emulator exists for the device of interest. With this strategy, the VMM traps and forwards the privileged I/O instructions and DMA requests issued by the guest OS device driver to the actual hardware. The programmer specifies which devices UML can access, and the VMM enforces the proper I/O port space and memory access for the device.

This second strategy requires extensions to enable ReVirt to log and replay the execution of the device driver. Whereas the first strategy placed the device emulator above the ReVirt logging layer, the second strategy forwards driver actions to the actual hardware device. Because this device may not be deterministic, ReVirt must log any information sent from the device to the driver. Specifically, ReVirt must log and replay the data returned

"Replay causes the virtual machine to transition through the same states as it went through during the original run"

"A VM can be replayed by starting from a checkpoint, then replaying [...] the network, keyboard, clock, and timing of interrupts"

Deterministic Replay

King et al. **Debugging Operating Systems with Time-Traveling Virtual Machines**

Usenix ‘05

6]. User-Mode Linux simulates interrupts and preemptions with asynchronous signals, and prior reverse debuggers are not able to replay such events. In addition, most reverse debuggers implement time travel by logging all changes to variables [30, 1, 21, 6], and this approach logs too much data when debugging long-running systems such as an OS. Finally, some systems work at the language level [27], and this prevents them from working with operating systems in a different language or with application binaries.

Researchers have worked to replay non-deterministic programs through various approaches. The events of different threads can be replayed at different levels, including logging accesses to shared objects [16], logging the scheduling order of multi-threaded programs on a uniprocessor [22], or logging physical memory accesses in hardware [2]. Other researchers have worked to optimize the amount of data logged [21].

Virtual-machine replay has been used for non-debugging purposes. Hypervisor used virtual-machine replay to synchronize the state of a backup machine to provide fault tolerance [5]. ReVirt used virtual-machine replay to enable detailed intrusion analysis [9]. Our work applies virtual-machine replay to achieve a new capability, which is reverse debugging of operating systems. TTVM also supports additional features over prior virtual-machine replay systems. TTVM supports the ability to run, log, and replay real device drivers in the guest OS, whereas prior virtual-machine replay systems ran only para-virtualized device drivers in the guest OS. In addition, TTVM can travel quickly forward and backward in time through its use of checkpoints and undo and redo logs, whereas ReVirt supported only a single checkpoint of a powered-off virtual machine and Hypervisor did not need to support time travel at all (it only supported replay within an epoch).

Another approach for providing time travel is to use a complete machine simulator, such as Simics [18]. Simics supports deterministic replay for operating systems and applications and has an interface to a debugger. However, Simics is drastically slower than TTVM, and this makes debugging long runs impractical. On a 750 MHz Ultrasparc III, Simics executes 2-6 million x86 instructions per second (several hundred times slower than native) [18], whereas virtual machines typically incur a slowdown of less than 2x.

8 Conclusions and future work

We have described the design and implementation of a time-traveling virtual machine and shown how to use TTVM to add powerful capabilities for debugging operating systems. We integrated TTVM with a general-purpose debugger, implementing commands such as reverse breakpoint, reverse watchpoint, and reverse step.

TTVM added reasonable overhead in the context of debugging. The logging needed to support time travel for three OS-intensive workloads added 3-12% in running time and 2-85 KB/sec in log space. Taking checkpoints every minute added less than 4% time overhead and 1-5 MB/sec space overhead. Taking checkpoints every 10 second to prepare for debugging a portion of a run added 16-33% overhead and enabled reverse debugging commands to complete in about 12 seconds.

We used TTVM and our new reverse debugging commands to fix four OS bugs that were difficult to find with standard debugging tools. We found the reverse debugging commands to be intuitive to understand and fast and easy to use. Reverse debugging proved especially helpful in finding bugs that were fragile due to non-determinism, bugs in device drivers, bugs that required long runs to trigger, bugs that corrupted the stack, and bugs that were detected after the relevant stack frame was popped.

Possible future work includes exploring non-traditional debugging operations that are enabled by time travel and deterministic replay. For example, one could measure the effects of a programmer-induced change by forking the execution and comparing the results after the change with the results of the original run.

9 Acknowledgments

Our shepherd, Steve Gribble, and the anonymous reviews provided feedback that helped improve this paper. This research was supported in part by ARDA grant NBCHC030104, National Science Foundation grants CCR-0098229 and CCR-0219085, and by Intel Corporation. Samuel King was supported in part by a National Defense Science and Engineering Graduate Fellowship.

References

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. An Execution-Backtracking Approach to Debugging. *IEEE Software*, 8(3), May 1991.

[2] D. F. Bacon and S. C. Goldstein. Hardware-Assisted Replay of Multiprocessor Programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1991.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 2003 Symposium on Operating Systems Principles*, October 2003.

[4] B. Boothe. Efficient algorithms for bidirectional debugging. In *Proceedings of the 2000 Conference on Programming Language Design and Implementation (PLDI)*, pages 299–310, June 2000.

“The logging added 3-12% in running time and 2-85 KB/sec in log space.”

“Taking checkpoints every minute added less than 4% time overhead and 1-5 MB/sec space overhead.”

“Taking checkpoints every 10 second added 16-33% overhead and enabled reverse debugging commands to complete in about 12 seconds”

- GDB: linked list storage of every memory op
  - \+ Easy to build and reason about
  - \- Both recording and replaying is very slow
- King (et al): Periodic state checkpointing
  - \+ Easy to jump between large period of time
  - \+ Can trade better fidelity for greater overhead
  - \- Have to replay execution between checkpoints
- King (et al): Only records external, non-deterministic events
  - \+ Reduces log size and improves performance
  - \- synchronizing external events becomes complicated

Improving performance with JIT compilation

compilation-based tooling

- Rather than interrupt execution to return control-flow to a debugger, weave instrumentation into the existing codebase at runtime
- Allows for both better performance and more flexible analysis tools

Luk et al. **Pin: Building Customized Program Analysis Tools w/ Dynamic Instrumentation**

PLDI ‘05

Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation

Chi-Keung Luk Robert Cohn Robert Muth Harish Patil Artur Klauser Geoff Lowney
Steven Wallace Vijay Janapa Reddi † Kim Hazelwood
Intel Corporation †University of Colorado
Website: http://rogue.colorado.edu/Pin, Email: pin.project@intel.com

Abstract

Robust and powerful software instrumentation tools are essential for program analysis tasks such as profiling, performance evaluation, and bug detection. To meet this need, we have developed a new instrumentation system called *Pin*. Our goals are to provide *easy-to-use*, *portable*, *transparent*, and *efficient* instrumentation. Instrumentation tools (called *Pintools*) are written in C/C++ using Pin's rich API. Pin follows the model of ATOM, allowing the tool writer to analyze an application at the instruction level without the need for detailed knowledge of the underlying instruction set. The API is designed to be *architecture independent* whenever possible, making Pintools source compatible across different architectures. However, a Pintool can access architecture-specific details when necessary. Instrumentation with Pin is mostly *transparent* as the application and Pintool observe the application's original, uninstrumented behavior. Pin uses *dynamic compilation* to instrument executables while they are running. For efficiency, Pin uses several techniques, including inlining, register re-allocation, liveness analysis, and instruction scheduling to optimize instrumentation. This fully automated approach delivers significantly better instrumentation performance than similar tools. For example, Pin is 3.3x faster than Valgrind and 2x faster than DynamoRIO for basic-block counting. To illustrate Pin's versatility, we describe two Pintools in daily use to analyze production software. Pin is publicly available for Linux platforms on four architectures: IA32 (32-bit x86), EM64T (64-bit x86), Itanium®, and ARM. In the ten months since Pin 2 was released in July 2004, there have been over 3000 downloads from its website.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging-code inspections and walk-throughs, debugging aids, tracing; D.3.4 [*Programming Languages*]: Processors-compilers, incremental compilers

General Terms Languages, Performance, Experimentation

Keywords Instrumentation, program analysis tools, dynamic compilation

PLDI'05 June 12–15,2005,Chicago,Illinois,USA.

1. Introduction

As software complexity increases, *instrumentation*—a technique for inserting extra code into an application to observe its behavior—is becoming more important. Instrumentation can be performed at various stages: in the source code, at compile time, post link time, or at run time. *Pin* is a software system that performs run-time binary instrumentation of Linux applications.

The goal of Pin is to provide an instrumentation platform for building a wide variety of program analysis tools for multiple architectures. As a result, the design emphasizes *ease-of-use*, *portability*, *transparency*, *efficiency*, and *robustness*. This paper describes the design of Pin and shows how it provides these features.

Pin's instrumentation is *easy to use*. Its user model is similar to the popular ATOM [30] API, which allows a tool to insert calls to instrumentation at arbitrary locations in the executable. Users do not need to manually inline instructions or save and restore state. Pin provides a rich API that abstracts away the underlying instruction set idiosyncrasies, making it possible to write *portable* instrumentation tools. The Pin distribution includes many sample architecture-independent *Pintools* including profilers, cache simulators, trace analyzers, and memory bug checkers. The API also allows access to architecture-specific information.

Pin provides *efficient* instrumentation by using a just-in-time (JIT) compiler to insert and optimize code. In addition to some standard techniques for dynamic instrumentation systems including code caching and trace linking, Pin implements *register reallocation*, *inlining*, *liveness analysis*, and *instruction scheduling* to optimize jitted code. This fully automated approach distinguishes Pin from most other instrumentation tools which require the user's assistance to boost performance. For example, Valgrind [22] relies on the tool writer to insert special operations in their intermediate representation in order to perform inlining; similarly DynamoRIO [6] requires the tool writer to manually inline and save/restore application registers.

Another feature that makes Pin efficient is *process attaching and detaching*. Like a debugger, Pin can attach to a process, instrument it, collect profiles, and eventually detach. The application only incurs instrumentation overhead during the period that Pin is attached. The ability to attach and detach is a necessity for the instrumentation of large, long-running applications.

Pin's JIT-based instrumentation defers code discovery until run time, allowing Pin to be more *robust* than systems that use static instrumentation or code patching. Pin can seamlessly handle mixed code and data, variable-length instructions, statically unknown indirect jump targets, dynamically loaded libraries, and dynamically generated code.

Pin preserves the original application behavior by providing instrumentation *transparency*. The application observes the same addresses (both instruction and data) and same values (both register

“The goal is to provide an *implementation platform* for building program analysis tools”

“Its API allows a tool to insert calls to instrumentation at arbitrary locations in the executable”

“Pin provides efficient instrumentation by using a just-in-time compiler to insert and optimize code”

...
4 byte write to **0x7fff572b3b1c**
4 byte write to **0x7fff572b3b1c**
1 byte write to **0x7f86c0c04b00**
1 byte write to **0x7f86c0c04b04**
4 byte write to **0x7fff572b3b20**
...

Luk et al. **Pin: Building Customized Program Analysis Tools w/ Dynamic Instrumentation**

PLDI ‘05

Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation

Chi-Keung Luk Rober
Steven

Website: http://rogue.colorado.edu/Pin, Email: pin.project@intel.com

Abstract

Robust and powerful software instrume
for program analysis tasks such as pro
ation, and bug detection. To meet this
a new instrumentation system called *P*
vide *easy-to-use*, *portable*, *transparent*,
tion. Instrumentation tools (called *Pinto*
using Pin's rich API. Pin follows the model of ATOM, allowing the
tool writer to analyze an application at
out the need for detailed knowledge of
set. The API is designed to be *architect*
possible, making Pintools source compat
tectures. However, a Pintool can access a
when necessary. Instrumentation with Pin is mostly *transparent* as
the application and Pintool observe the application's original, uninstrumented behavior. Pin uses *dynamic compilation* to instrument executables while they are running. For efficiency, Pin uses several techniques, including inlining, register re-allocation, liveness analysis, and instruction scheduling to optimize instrumentation. This fully automated approach delivers significantly better instrumentation performance than similar tools. For example, Pin is 3.3x faster than Valgrind and 2x faster than DynamoRIO for basic-block counting. To illustrate Pin's versatility, we describe two Pintools in daily use to analyze production software. Pin is publicly available for Linux platforms on four architectures: IA32 (32-bit x86), EM64T (64-bit x86), Itanium®, and ARM. In the ten months since Pin 2 was released in July 2004, there have been over 3000 downloads from its website.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging-code inspections and walk-throughs, debugging aids, tracing; D.3.4 [*Programming Languages*]: Processors-compilers, incremental compilers

General Terms Languages, Performance, Experimentation

Keywords Instrumentation, program analysis tools, dynamic compilation

PLDI'05 June 12–15,2005,Chicago,Illinois,USA.

The goal of Pin is to provide an instrumentation platform for

to the popular ATOM [30] API, which allows a tool to insert calls to instrumentation at arbitrary locations in the executable. Users do not need to manually inline instructions or save and restore state. Pin provides a rich API that abstracts away the underlying instruction set idiosyncrasies, making it possible to write *portable* instrumentation tools. The Pin distribution includes many sample architecture-independent *Pintools* including profilers, cache simulators, trace analyzers, and memory bug checkers. The API also allows access to architecture-specific information.

Pin provides *efficient* instrumentation by using a just-in-time (JIT) compiler to insert and optimize code. In addition to some standard techniques for dynamic instrumentation systems including code caching and trace linking, Pin implements *register reallocation*, *inlining*, *liveness analysis*, and *instruction scheduling* to optimize jitted code. This fully automated approach distinguishes Pin from most other instrumentation tools which require the user's assistance to boost performance. For example, Valgrind [22] relies on the tool writer to insert special operations in their intermediate representation in order to perform inlining; similarly DynamoRIO [6] requires the tool writer to manually inline and save/restore application registers.

Another feature that makes Pin efficient is *process attaching and detaching*. Like a debugger, Pin can attach to a process, instrument it, collect profiles, and eventually detach. The application only incurs instrumentation overhead during the period that Pin is attached. The ability to attach and detach is a necessity for the instrumentation of large, long-running applications.

Pin's JIT-based instrumentation defers code discovery until run time, allowing Pin to be more *robust* than systems that use static instrumentation or code patching. Pin can seamlessly handle mixed code and data, variable-length instructions, statically unknown indirect jump targets, dynamically loaded libraries, and dynamically generated code.

Pin preserves the original application behavior by providing instrumentation *transparency*. The application observes the same ad-

dresses (both instruction and data) and same values (both register

```
recordMemWrite(%rip, &i);
recordMemWrite(%rip, &i);
recordMemWrite(%rip, &j);
recordMemWrite(%rip, &i);
```

```
8 void bar() {
  int i = 7;
  i++;
  int j = i - 3;
  i = i + j;
```

INS_AddInstrumentFunction()

```
movl    $0x7, -0x8(%rbp)
```

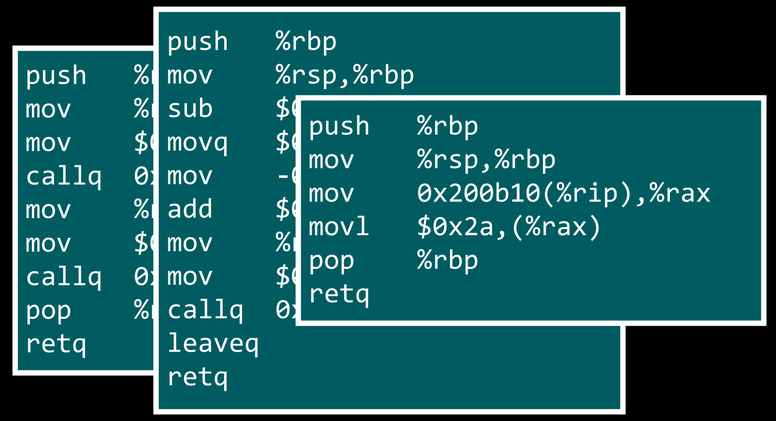
TRACE_AddInstrumentFunction()

```
push   %rbp
mov    %rsp,%rbp
mov    $0x100,%edi
callq  0x400430 <malloc@plt>
```

RTN_AddInstrumentFunction()

```
push   %rbp
mov    %rsp,%rbp
mov    $0x100,%edi
callq  0x400430 <malloc@plt>
mov    %rax,0x200ac8(%rip)
mov    $0x0,%eax
callq  0x400540 <bar>
pop    %rbp
retq
```

IMG_AddInstrumentFunction()



```
static void
instrument_gimple (gimple_stmt_iterator gsi)
{
  unsigned i;
  gimple stmt;
  enum gimple_code gcode;
  tree rhs, lhs;

  stmt = gsi_stmt (gsi);
  gcode = gimple_code (stmt);
  if (gcode == GIMPLE_ASSIGN) {
      /* Handle assignment lhs as store.  */
      lhs = gimple_assign_lhs (stmt);
      instrument_expr (gsi, lhs, 1);
      /* Handle operands as loads.  */
      for (i = 1; i < gimple_num_ops (stmt); i++) {
          rhs = gimple_op (stmt, i);
          instrument_expr (gsi, rhs, 0);
        }
    }
}
```

```
static void
instrument_expr (gimple_stmt_iterator gsi, tree expr, int is_write)
{
  enum tree_code tcode;
  unsigned fld_off, fld_size;
  tree base, rhs;
  gimple stmt;
  gimple_seq gs;
  location_t loc;

  base = get_base_address (expr);
  if (base == NULL_TREE || TREE_CODE (base) == SSA_NAME
      || TREE_CODE (base) == STRING_CST)
    return;

  tcode = TREE_CODE (expr);

  /* Below are things we do not instrument
     (no possibility of races or not implemented yet).  */
  if (/* Compiler-emitted artificial variables.  */
      (DECL_P (expr) && DECL_ARTIFICIAL (expr))
      /* The var does not live in memory -> no possibility of races.  */
      || (tcode == VAR_DECL
          && TREE_ADDRESSABLE (expr) == 0
          && TREE_STATIC (expr) == 0)
      /* Not implemented.  */
      || TREE_CODE (TREE_TYPE (expr)) == RECORD_TYPE
      || tcode == CONSTRUCTOR
      || tcode == PARM_DECL
      /* Load of a const variable/parameter/field.  */
      || is_load_of_const (expr, is_write))
    return;
```

```
  if (tcode == COMPONENT_REF) {
      tree field = TREE_OPERAND (expr, 1);
      if (TREE_CODE (field) == FIELD_DECL) {
          fld_off = TREE_INT_CST_LOW (DECL_FIELD_BIT_OFFSET (field));
          fld_size = TREE_INT_CST_LOW (DECL_SIZE (field));
          if (((fld_off % BITS_PER_UNIT) != 0)
              || ((fld_size % BITS_PER_UNIT) != 0)) {
              /* As of now it crashes compilation.
                 TODO: handle bit-fields as if touching the whole field.  */
              return;
            }
        }
    }

  /* TODO: handle other cases
     (FIELD_DECL, MEM_REF, ARRAY_RANGE_REF, TARGET_MEM_REF, ADDR_EXPR).  */
  if (tcode != ARRAY_REF
      && tcode != VAR_DECL && tcode != COMPONENT_REF
      && tcode != INDIRECT_REF && tcode != MEM_REF)
    return;

  func_mops++;
  stmt = gsi_stmt (gsi);
  loc = gimple_location (stmt);
  rhs = is_vptr_store (stmt, expr, is_write);
  if (rhs == NULL)
    gs = instr_memory_access (expr, is_write);
  else
    gs = instr_vptr_update (expr, rhs);
  set_location (gs, loc);
  /* Instrumentation for assignment of a function result
     must be inserted after the call.  Instrumentation for
     reads of function arguments must be inserted before the call.
     That's because the call can contain synchronization.  */
  if (is_gimple_call (stmt) && is_write)
    gsi_insert_seq_after (&gsi, gs, GSI_NEW_STMT);
  else
    gsi_insert_seq_before (&gsi, gs, GSI_SAME_STMT);
}
```

```
static gimple_seq
instr_memory_access (tree expr, int is_write)
{
  tree addr_expr, expr_type, call_expr, fdecl;
  gimple_seq gs;
  unsigned size;

  gcc_assert (is_gimple_addressable (expr));
  addr_expr = build_addr (unshare_expr (expr), current_function_decl);
  expr_type = TREE_TYPE (expr);
  while (TREE_CODE (expr_type) == ARRAY_TYPE)
    expr_type = TREE_TYPE (expr_type);
  size = (TREE_INT_CST_LOW (TYPE_SIZE (expr_type))) / BITS_PER_UNIT;
  fdecl = get_memory_access_decl (is_write, size);
  call_expr = build_call_expr (fdecl, 1, addr_expr);
  gs = NULL;
  force_gimple_operand (call_expr, &gs, true, 0);
  return gs;
```

```
static tree
get_memory_access_decl (int is_write, unsigned size)
{
  tree typ, *decl;
  char fname [64];
  static tree cache [2][17];

  is_write = !!is_write;
  if (size <= 1)
    size = 1;
  else if (size <= 3)
    size = 2;
  else if (size <= 7)
    size = 4;
  else if (size <= 15)
    size = 8;
  else
    size = 16;
  decl = &cache[is_write][size];
  if (*decl == NULL) {
      snprintf(fname, sizeof fname, "__tsan_%s%d",
               is_write ? "write" : "read", size);
      typ = build_function_type_list (void_type_node,
                  ptr_type_node, NULL_TREE);
      *decl = build_func_decl (typ, fname);
    }
  return *decl;
}
```

thread-sanitizer/blob/master/gcc/tree-tsan.c

Luk et al. **Pin: Building Customized Program Analysis Tools w/ Dynamic Instrumentation**

PLDI '05

[1] Although EM64T is a 64-bit extension of IA32, we classify it as a separate architecture because of its many new features such as 64-bit addressing, a flat address space, twice the number of registers, and new software conventions [15].

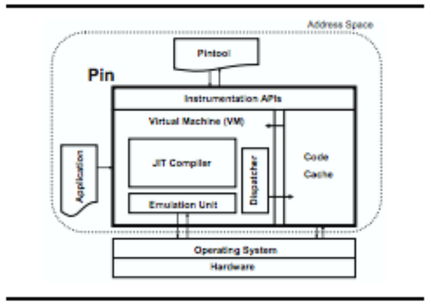


Figure 2. Pin's software architecture

mentation API invoked by Pintools. The VM consists of a just-in-time compiler (JIT), an emulator, and a dispatcher. After Pin gains control of the application, the VM coordinates its components to execute the application. The JIT compiles and instruments application code, which is then launched by the dispatcher. The compiled code is stored in the code cache. Entering/leaving the VM from/to the code cache involves saving and restoring the application register state. The emulator interprets instructions that cannot be executed directly. It is used for system calls which require special handling from the VM. Since Pin sits above the operating system, it can only capture user-level code.

As Figure 2 shows, there are three binary programs present when an instrumented program is running: the application, Pin, and the Pintool. Pin is the engine that jits and instruments the application. The Pintool contains the instrumentation and analysis routines and is linked with a library that allows it to communicate with Pin. While they share the *same* address space, they do *not* share any libraries and so there are typically three copies of `glibc`. By making all of the libraries private, we avoid unwanted interaction between Pin, the Pintool, and the application. One example of a problematic interaction is when the application executes a `glibc` function that is not reentrant. If the application starts executing the function and then tries to execute some code that triggers further compilation, it will enter the JIT. If the JIT executes the same `glibc` function, it will enter the same procedure a second time while the application is still executing it, causing an error. Since we have separate copies of `glibc` for each component, Pin and the application do not share any data and cannot have a re-entrancy problem. The same problem can occur when we jit the analysis code in the Pintool that calls `glibc` (jitting the analysis routine allows us to greatly reduce the overhead of simple instrumentation on Itanium).

3.2 Injecting Pin

The injector loads Pin into the address space of an application. Injection uses the Unix Ptrace API to obtain control of an application and capture the processor context. It loads the Pin binary into the application address space and starts it running. After initializing itself, Pin loads the Pintool into the address space and starts it running. The Pintool initializes itself and then requests that Pin start the application. Pin creates the initial context and starts jitting the application at the entry point (or at the current PC in the case of attach). Using Ptrace as the mechanism for injection allows us to attach to an already running process in the same way as a debugger. It is also possible to detach from an instrumented process and continue executing the original, uninstrumented code.

3.1 System Overview

Figure 2 illustrates Pin's software architecture. At the highest level, Pin consists of a virtual machine (VM), a code cache, and an instru-

Other tools like DynamoRIO [6] rely on the `LD_PRELOAD` environment variable to force the dynamic loader to load a shared library in the address space. First, `LD_PRELOAD` does not work with *statically-linked* binaries, which many of our users require. Second, loading an extra shared library will shift all of the application shared libraries and some dynamically allocated memory to a higher address when compared to an uninstrumented execution. We attempt to preserve the original behavior as much as possible. Third, the instrumentation tool cannot gain control of the application until after the shared-library loader has partially executed, while our method is able to instrument the very first instruction in the program. This capability actually exposed a bug in the Linux shared-library loader, resulting from a reference to uninitialized data on the stack.

3.3 The JIT Compiler

3.3.1 Basics

Pin compiles from one ISA directly into the same ISA (e.g., IA32 to IA32, ARM to ARM) without going through an intermediate format, and the compiled code is stored in a software-based code cache. Only code residing in the code cache is executed—the original code is never executed. An application is compiled one *trace* at a time. A trace is a straight-line sequence of instructions which terminates at one of the conditions: (i) an *unconditional* control transfer (branch, call, or return), (ii) a pre-defined number of *conditional* control transfers, or (iii) a pre-defined number of instructions have been fetched in the trace. In addition to the last exit, a trace may have multiple side-exits (the conditional control transfers). Each exit initially branches to a *stub*, which re-directs the control to the VM. The VM determines the target address (which is statically unknown for indirect control transfers), generates a new trace for the target if it has not been generated before, and resumes the execution at the target trace.

In the rest of this section, we discuss the following features of our JIT: trace linking, register re-reallocation, and instrumentation optimization. Our current performance effort is focusing on IA32, EM64T, and Itanium, which have all these features implemented. While the ARM version of Pin is fully functional, some of the optimizations are not yet implemented.

3.3.2 Trace Linking

To improve performance, Pin attempts to branch directly from a trace exit to the target trace, bypassing the stub and VM. We call this process *trace linking*. Linking a *direct* control transfer is straightforward as it has a unique target. We simply patch the branch at the end of one trace to jump to the target trace. However, an *indirect* control transfer (a jump, call, or return) has multiple possible targets and therefore needs some sort of target-prediction mechanism.

Figure 3(a) illustrates our indirect linking approach as implemented on the x86 architecture. Pin translates the indirect jump into a move and a direct jump. The move puts the indirect target address into register `%edx` (this register as well as the `%ecx` and `%esi` shown in Figure 3(a) are obtained via register re-allocation, as we will discuss in Section 3.3.3). The direct jump goes to the first predicted target address `0x40001000` (which is mapped to `0x70001000` in the code cache for this example). We compare `%edx` against `0x40001000` using the `lea/jecxz` idiom used in DynamoRIO [6], which avoids modifying the conditional flags register `eflags`. If the prediction is correct (i.e. `%ecx=0`), we will branch to `match1` to execute the remaining code of the predicted target. If the prediction is wrong, we will try another predicted target `0x40002000` (mapped to `0x70002000` in the code cache). If the target is not found on the chain, we will branch to `Lookup1tab_1`, which searches for the target in a hash table (whose base address is

"Pin compiles from one ISA directly into the same ISA without going through an intermediate format and stored in a software-based code cache"

"only code residing in the code cache is executed - the original code is never executed."

Luk et al. **Pin: Building Customized Program Analysis Tools w/ Dynamic Instrumentation**

PLDI ‘05

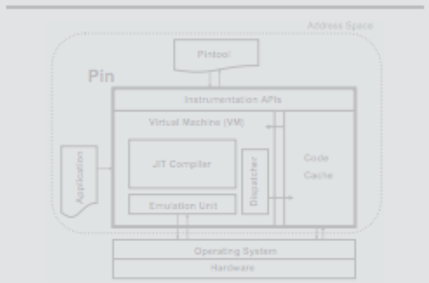


Figure 2. Pin’s software architecture

“Pin compiles from one ISA directly into the same ISA without going through an intermediate format”

```
movl    $0x7, -0x8(%rbp)
```

```
addl    $0x1, -0x8(%rbp)
mov     -0x8(%rbp), %eax
sub     $0x3, %eax
```

```
mov     %eax, -0x4(%rbp)
mov     $0x4(%rbp), %eax
```

```
add     %eax, -0x8(%rbp)
```

```
recordMemWrite(%rip, -0x8(%rbp))
movl    $0x7, -0x8(%rbp)
```

```
recordMemWrite(%rip, -0x8(%rbp))
addl    $0x1, -0x8(%rbp)
```

```
mov     -0x8(%rbp), %eax
sub     $0x3, %eax
mov     %eax, -0x4(%rbp)
```

```
recordMemWrite(%rip, -0x4(%rbp))
mov     $0x4(%rbp), %eax
```

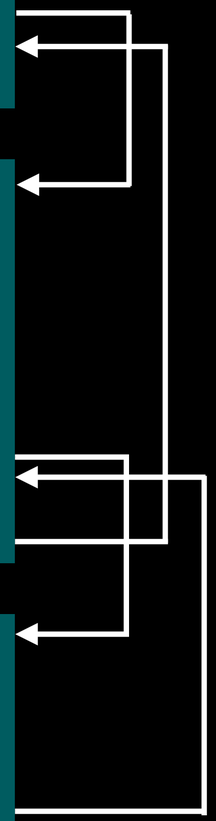
```
recordMemWrite(%rip, -0x8(%rbp))
add     %eax, -0x8(%rbp)
```

code rewriting

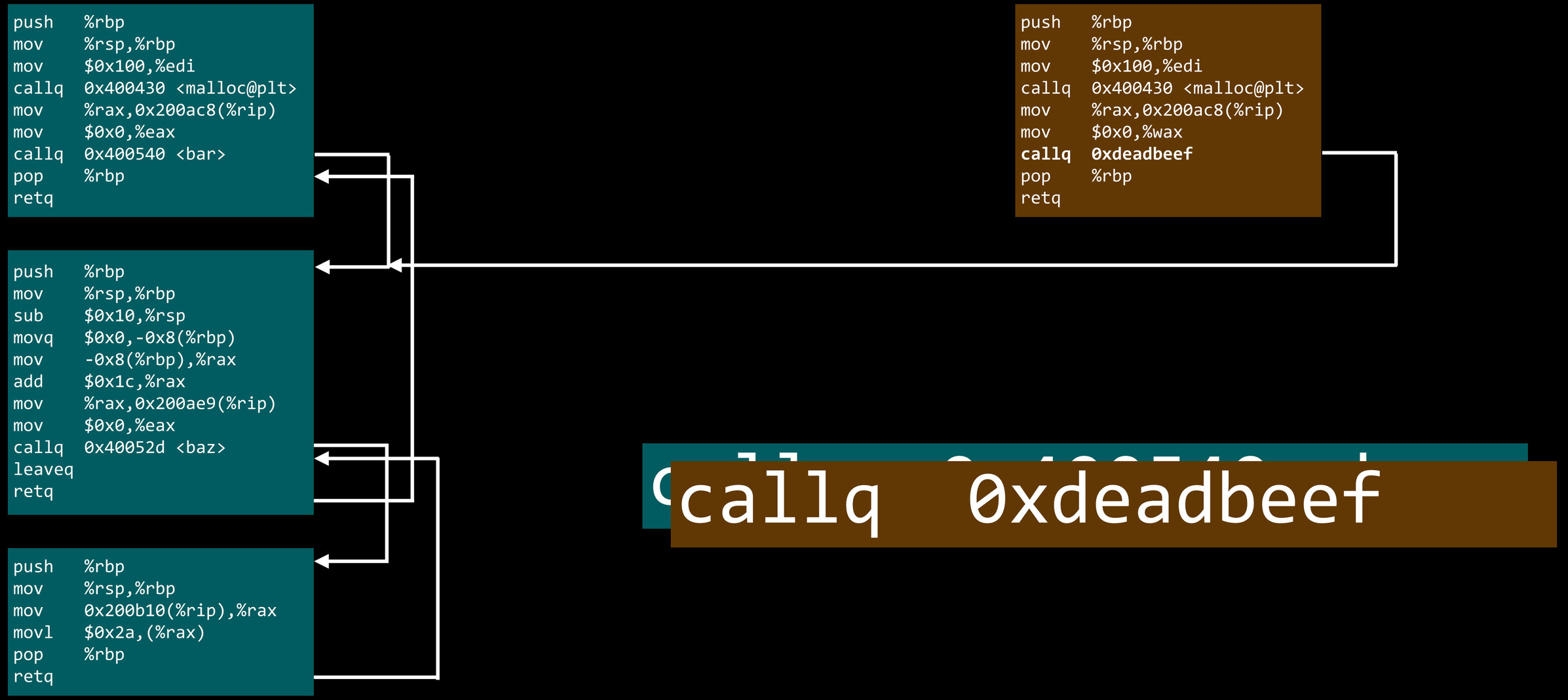
```
push   %rbp
mov    %rsp,%rbp
mov    $0x100,%edi
callq  0x400430 <malloc@plt>
mov    %rax,0x200ac8(%rip)
mov    $0x0,%eax
callq  0x400540 <bar>
pop    %rbp
retq
```

```
push   %rbp
mov    %rsp,%rbp
sub    $0x10,%rsp
movq   $0x0,-0x8(%rbp)
mov    -0x8(%rbp),%rax
add    $0x1c,%rax
mov    %rax,0x200ae9(%rip)
mov    $0x0,%eax
callq  0x40052d <baz>
leaveq
retq
```

```
push   %rbp
mov    %rsp,%rbp
mov    0x200b10(%rip),%rax
movl   $0x2a,(%rax)
pop    %rbp
retq
```

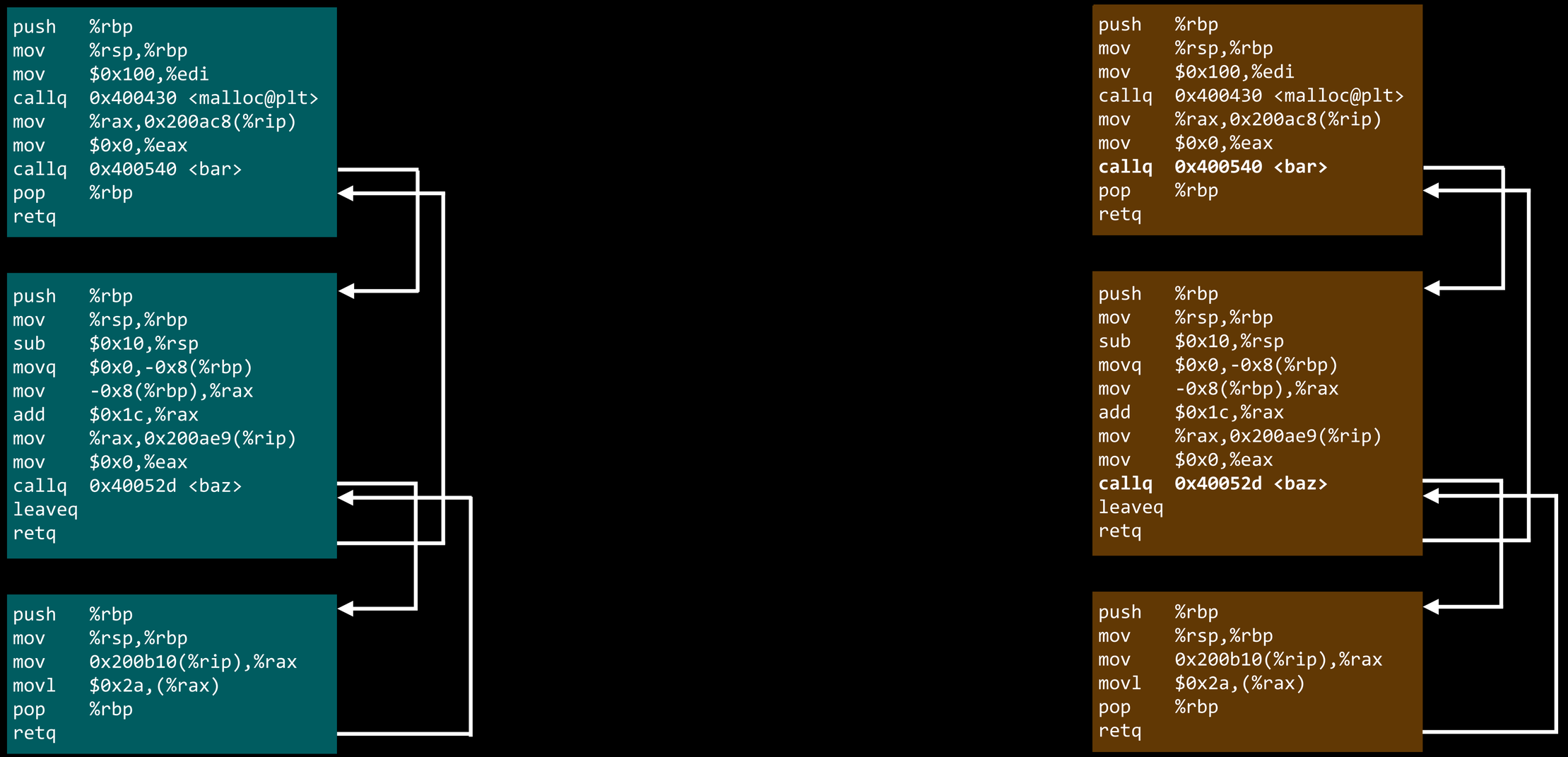


jump rewriting



jump rewriting

jump rewriting



stateless(ish) instrumentation

```
// This routine is executed each time malloc is called.
VOID BeforeMalloc( int size, THREADID threadid )
{
    PIN_GetLock(&lock, threadid+1);
    fprintf(out, "thread %d entered malloc(%d)\n", threadid, size);
    fflush(out);
    PIN_ReleaseLock(&lock);
}
```

```
// This function is called before every instruction is executed
VOID docount() { icount++; }
```

https://software.intel.com/sites/landingpage/pintool/docs/67254/Pin/html/index.html#EXAMPLES

Nethercote et al. **Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation**

PLDI '07

Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation

Nicholas Nethercote
National ICT Australia, Melbourne, Australia
njn@csse.unimelb.edu.au

Julian Seward
OpenWorks LLP, Cambridge, UK
julian@open-works.co.uk

Abstract

Dynamic binary instrumentation (DBI) frameworks make it easy to build dynamic binary analysis (DBA) tools such as checkers and profilers. Much of the focus on DBI frameworks has been on performance; little attention has been paid to their capabilities. As a result, we believe the potential of DBI has not been fully exploited.

In this paper we describe Valgrind, a DBI framework designed for building heavyweight DBA tools. We focus on its unique support for *shadow values*—a powerful but previously little-studied and difficult-to-implement DBA technique, which requires a tool to shadow every register and memory value with another value that describes it. This support accounts for several crucial design features that distinguish Valgrind from other DBI frameworks. Because of these features, lightweight tools built with Valgrind run comparatively slowly, but Valgrind can be used to build more interesting, heavyweight tools that are difficult or impossible to build with other DBI frameworks such as Pin and DynamoRIO.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—debugging aids, monitors; D.3.4 [*Programming Languages*]: Processors—incremental compilers

General Terms Design, Performance, Experimentation

Keywords Valgrind, Memcheck, dynamic binary instrumentation, dynamic binary analysis, shadow values

1. Introduction

Valgrind is a dynamic binary instrumentation (DBI) framework that occupies a unique part of the DBI framework design space. This paper describes how it works, and how it differs from other frameworks.

1.1 Dynamic Binary Analysis and Instrumentation

Many programmers use program analysis tools, such as error checkers and profilers, to improve the quality of their software. *Dynamic binary analysis* (DBA) tools are one such class of tools; they analyse programs at run-time at the level of machine code.

DBA tools are often implemented using *dynamic binary instrumentation* (DBI), whereby the *analysis code* is added to the original code of the *client program* at run-time. This is convenient for users, as no preparation (such as recompiling or relinking) is needed. Also, it gives 100% instrumentation coverage of user-mode code, without requiring source code. Several generic *DBI frameworks* exist, such as Pin [11], DynamoRIO [3], and Valgrind [18, 15]. They provide a base system that can instrument and run code, plus an environment for writing tools that plug into the base system.

The performance of DBI frameworks has been studied closely [1, 2, 9]. Less attention has been paid to their instrumentation capabilities, and the tools built with them. This is a shame, as it is the tools that make DBI frameworks useful, and complex tools are more interesting than simple tools. As a result, we believe the potential of DBI has not been fully exploited.

1.2 Shadow Value Tools and Heavyweight DBA

One interesting group of DBA tools are those that use *shadow values*. These tools shadow, purely in software, every register and memory value with another value that says something about it. We call these *shadow value tools*. Consider the following motivating list of shadow value tools; the descriptions are brief but demonstrate that shadow values (a) can be used in a wide variety of ways, and (b) are powerful and interesting.

Memcheck [25] uses shadow values to track which bit values are undefined (i.e. uninitialised, or derived from undefined values) and can thus detect dangerous uses of undefined values. It is used by thousands of C and C++ programmers, and is probably the most widely-used DBA tool in existence.[1]

TaintCheck [20] tracks which byte values are tainted (i.e. from an untrusted source, or derived from tainted values) and can thus detect dangerous uses of tainted values. *TaintTrace* [6] and *LIFT* [23] are similar tools.

McCamant and Ernst's secret-tracking tool [13] tracks which bit values are secret (e.g. passwords), and determines how much information about secret inputs is revealed by public outputs.

Hobbes [4] tracks each value's type (determined from operations performed on the value) and can thus detect subsequent operations inappropriate for a value of that type.

DynCompB [7] similarly determines abstract types of byte values, for program comprehension and invariant detection purposes.

Annelid [16] tracks which word values are array pointers, and from this can detect bounds errors.

Redux [17] creates a *dynamic dataflow graph*, a visualisation of a program's entire computation; from the graph one can see all the prior operations that contributed to the each value's creation.

In these tools each shadow value records a simple approximation of each value's history—e.g. one shadow bit per bit, one

PLDI'07 June 11–13, 2007, San Diego, California, USA.

[1] Purify [8] is a memory-checking tool similar to Memcheck. However, Purify is not a shadow value tool as it does not does not track definedness of values through registers. As a result, it detects undefined value errors less accurately than Memcheck.

"which values are undefined (i.e. uninitialised, or derived from undefined values) and can thus detect dangerous uses of undefined values."

"which values are tainted (i.e. from an untrusted source,) and can thus detect dangerous uses of tainted values."

"which word values are array pointers, and from this can detect bounds errors."

Nethercote et al. **Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation**

PLDI '07

Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation

Nicholas Nethercote
National ICT Australia, Melbourne, Australia
njn@csse.unimelb.edu.au

Julian Seward
OpenWorks LLP, Cambridge, UK
julian@open-works.co.uk

Abstract

Dynamic binary instrumentation (DBI) frameworks make it easy to build dynamic binary analysis (DBA) tools such as checkers and profilers. Much of the focus on DBI frameworks has been on performance; little attention has been paid to their capabilities. As a result, we believe the potential of DBI has not been fully exploited.

In this paper we describe Valgrind, a DBI framework designed for building heavyweight DBA tools. We focus on its unique support for *shadow values*—a powerful but previously little-studied and difficult-to-implement DBA technique, which requires a tool to shadow every register and memory value with another value that describes it. This support accounts for several crucial design features that distinguish Valgrind from other DBI frameworks. Because of these features, lightweight tools built with Valgrind run comparatively slowly, but Valgrind can be used to build more interesting, heavyweight tools that are difficult or impossible to build with other DBI frameworks such as Pin and DynamoRIO.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—debugging aids, monitors; D.3.4 [*Programming Languages*]: Processors—incremental compilers

General Terms Design, Performance, Experimentation

Keywords Valgrind, Memcheck, dynamic binary instrumentation, dynamic binary analysis, shadow values

1. Introduction

Valgrind is a dynamic binary instrumentation (DBI) framework that occupies a unique part of the DBI framework design space. This paper describes how it works, and how it differs from other frameworks.

1.1 Dynamic Binary Analysis and Instrumentation

Many programmers use program analysis tools, such as error checkers and profilers, to improve the quality of their software. *Dynamic binary analysis* (DBA) tools are one such class of tools; they analyse programs at run-time at the level of machine code.

DBA tools are often implemented using *dynamic binary instrumentation* (DBI), whereby the *analysis code* is added to the original code of the *client program* at run-time. This is convenient for users, as no preparation (such as recompiling or relinking) is needed. Also, it gives 100% instrumentation coverage of user-mode code, without requiring source code. Several generic *DBI frameworks* exist, such as Pin [11], DynamoRIO [3], and Valgrind [18, 15]. They provide a base system that can instrument and run code, plus an environment for writing tools that plug into the base system.

The performance of DBI frameworks has been studied closely [1, 2, 9]. Less attention has been paid to their instrumentation capabilities, and the tools built with them. This is a shame, as it is the tools that make DBI frameworks useful, and complex tools are more interesting than simple tools. As a result, we believe the potential of DBI has not been fully exploited.

1.2 Shadow Value Tools and Heavyweight DBA

One interesting group of DBA tools are those that use *shadow values*. These tools shadow, purely in software, every register and memory value with another value that says something about it. We call these *shadow value tools*. Consider the following motivating list of shadow value tools; the descriptions are brief but demonstrate that shadow values (a) can be used in a wide variety of ways, and (b) are powerful and interesting.

Memcheck [25] uses shadow values to track which bit values are undefined (i.e. uninitialised, or derived from undefined values) and can thus detect dangerous uses of undefined values. It is used by thousands of C and C++ programmers, and is probably the most widely-used DBA tool in existence.[1]

TaintCheck [20] tracks which byte values are tainted (i.e. from an untrusted source, or derived from tainted values) and can thus detect dangerous uses of tainted values. *TaintTrace* [6] and *LIFT* [23] are similar tools.

McCamant and Ernst's secret-tracking tool [13] tracks which bit values are secret (e.g. passwords), and determines how much information about secret inputs is revealed by public outputs.

Hobbes [4] tracks each value's type (determined from operations performed on the value) and can thus detect subsequent operations inappropriate for a value of that type.

DynCompB [7] similarly determines abstract types of byte values, for program comprehension and invariant detection purposes.

Annelid [16] tracks which word values are array pointers, and from this can detect bounds errors.

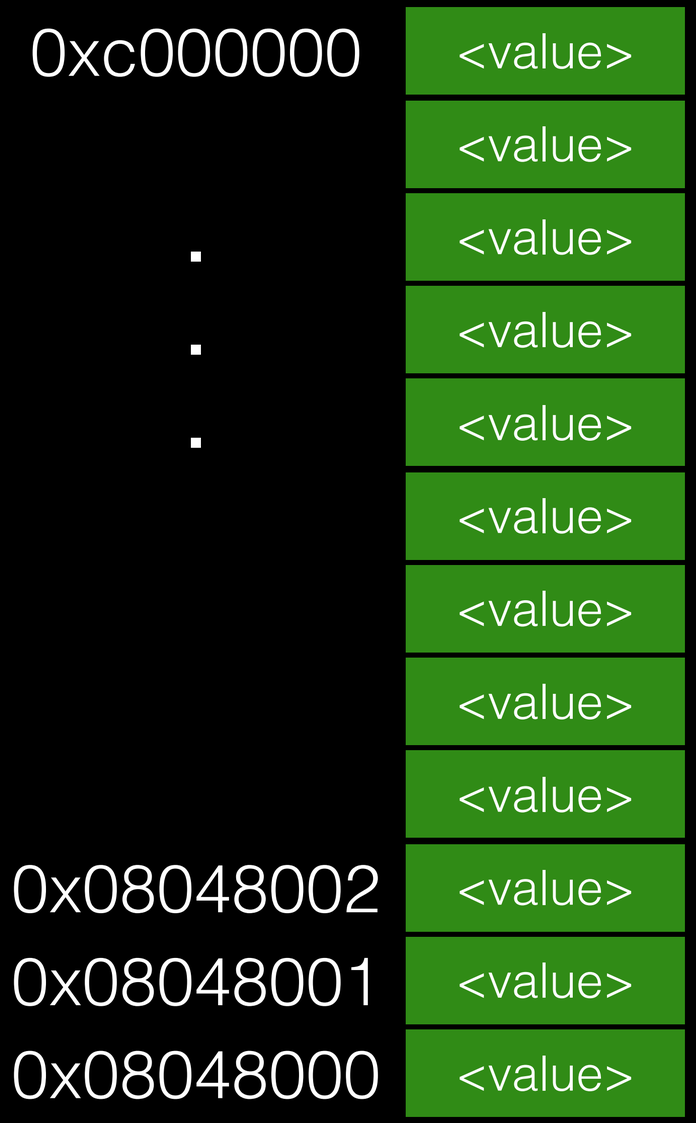
Redux [17] creates a *dynamic dataflow graph*, a visualisation of a program's entire computation; from the graph one can see all the prior operations that contributed to the each value's creation.

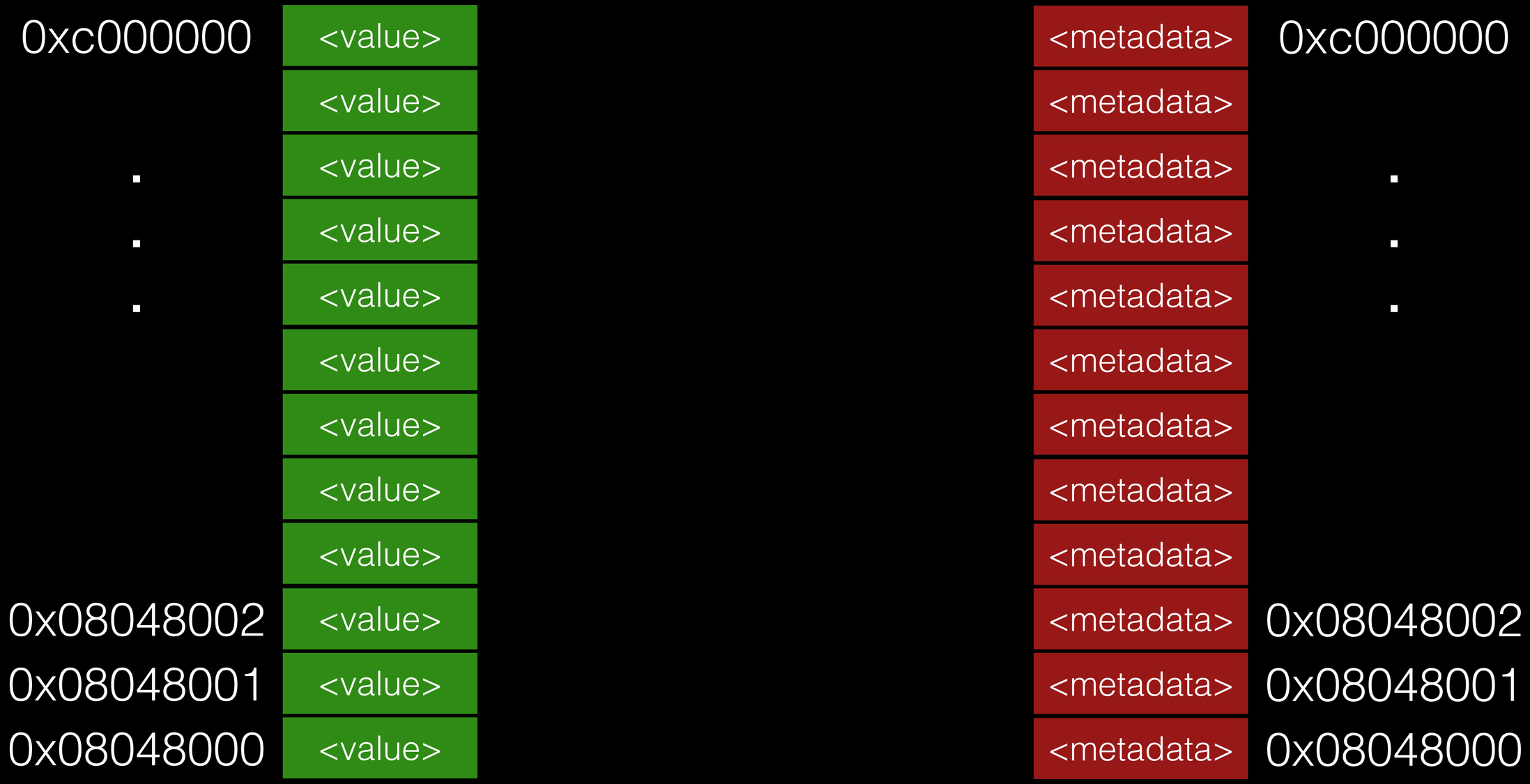
In these tools each shadow value records a simple approximation of each value's history—e.g. one shadow bit per bit, one

PLDI'07 June 11–13, 2007, San Diego, California, USA.

[1] Purify [8] is a memory-checking tool similar to Memcheck. However, Purify is not a shadow value tool as it does not does not track definedness of values through registers. As a result, it detects undefined value errors less accurately than Memcheck.

One interesting group of DBA tools are those that use shadow values. These tools shadow, purely in software, every register and memory value with another value that says something about it. We call these shadow value tools.


0xc000000
<value>
<value>
<value>
<value>
<value>
<value>
<value>
<value>
<value>
<value>
0x08048002
<value>
0x08048001
<value>
0x08048000
<value>


0xc000000
<value>
<value>
<value>
<value>
<value>
<value>
<value>
<value>
<value>
<value>
0x08048002
0x08048001
0x08048000
<metadata>
<metadata>
<metadata>
<metadata>
<metadata>
<metadata>
<metadata>
<metadata>
<metadata>
<metadata>
0xc000000
0x08048002
0x08048001
0x08048000

Nethercote et al. **Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation**

PLDI '07

Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation

Nicholas Nethercote
National ICT Australia, Melbourne, Australia
njn@csse.unimelb.edu.au

Julian Seward
OpenWorks LLP, Cambridge, UK
julian@open-works.co.uk

Abstract

Dynamic binary instrumentation (DBI) frameworks make it easy to build dynamic binary analysis (DBA) tools such as checkers and profilers. Much of the focus on DBI frameworks has been on performance; little attention has been paid to their capabilities. As a result, we believe the potential of DBI has not been fully exploited.

In this paper we describe Valgrind, a DBI framework designed for building heavyweight DBA tools. We focus on its unique support for *shadow values*—a powerful but previously little-studied and difficult-to-implement DBA technique, which requires a tool to shadow every register and memory value with another value that describes it. This support accounts for several crucial design features that distinguish Valgrind from other DBI frameworks. Because of these features, lightweight tools built with Valgrind run comparatively slowly, but Valgrind can be used to build more interesting, heavyweight tools that are difficult or impossible to build with other DBI frameworks such as Pin and DynamoRIO.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—debugging aids, monitors; D.3.4 [*Programming Languages*]: Processors—incremental compilers

General Terms Design, Performance, Experimentation

Keywords Valgrind, Memcheck, dynamic binary instrumentation, dynamic binary analysis, shadow values

1. Introduction

Valgrind is a dynamic binary instrumentation (DBI) framework that occupies a unique part of the DBI framework design space. This paper describes how it works, and how it differs from other frameworks.

1.1 Dynamic Binary Analysis and Instrumentation

Many programmers use program analysis tools, such as error checkers and profilers, to improve the quality of their software. *Dynamic binary analysis* (DBA) tools are one such class of tools; they analyse programs at run-time at the level of machine code.

DBA tools are often implemented using *dynamic binary instrumentation* (DBI), whereby the *analysis code* is added to the original code of the *client program* at run-time. This is convenient for users, as no preparation (such as recompiling or relinking) is needed. Also, it gives 100% instrumentation coverage of user-mode code, without requiring source code. Several generic *DBI frameworks* exist, such as Pin [11], DynamoRIO [3], and Valgrind [18, 15]. They provide a base system that can instrument and run code, plus an environment for writing tools that plug into the base system.

The performance of DBI frameworks has been studied closely [1, 2, 9]. Less attention has been paid to their instrumentation capabilities, and the tools built with them. This is a shame, as it is the tools that make DBI frameworks useful, and complex tools are more interesting than simple tools. As a result, we believe the potential of DBI has not been fully exploited.

1.2 Shadow Value Tools and Heavyweight DBA

One interesting group of DBA tools are those that use *shadow values*. These tools shadow, purely in software, every register and memory value with another value that says something about it. We call these *shadow value tools*. Consider the following motivating list of shadow value tools; the descriptions are brief but demonstrate that shadow values (a) can be used in a wide variety of ways, and (b) are powerful and interesting.

Memcheck [25] uses shadow values to track which bit values are undefined (i.e. uninitialised, or derived from undefined values) and can thus detect dangerous uses of undefined values. It is used by thousands of C and C++ programmers, and is probably the most widely-used DBA tool in existence.[1]

TaintCheck [20] tracks which byte values are tainted (i.e. from an untrusted source, or derived from tainted values) and can thus detect dangerous uses of tainted values. *TaintTrace* [6] and *LIFT* [23] are similar tools.

McCamant and Ernst's secret-tracking tool [13] tracks which bit values are secret (e.g. passwords), and determines how much information about secret inputs is revealed by public outputs.

Hobbes [4] tracks each value's type (determined from operations performed on the value) and can thus detect subsequent operations inappropriate for a value of that type.

DynCompB [7] similarly determines abstract types of byte values, for program comprehension and invariant detection purposes.

Annelid [16] tracks which word values are array pointers, and from this can detect bounds errors.

Redux [17] creates a *dynamic dataflow graph*, a visualisation of a program's entire computation; from the graph one can see all the prior operations that contributed to the each value's creation.

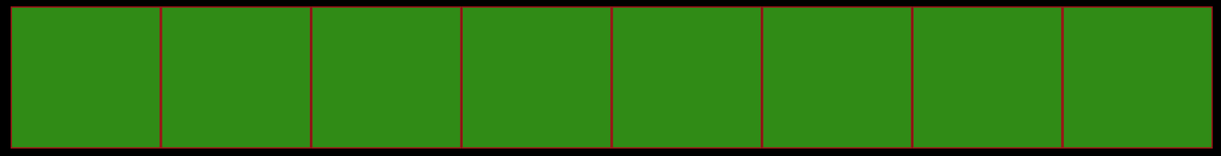
In these tools each shadow value records a simple approximation of each value's history—e.g. one shadow bit per bit, one

PLDI'07 June 11–13, 2007, San Diego, California, USA.

[1] Purify [8] is a memory-checking tool similar to Memcheck. However, Purify is not a shadow value tool as it does not does not track definedness of values through registers. As a result, it detects undefined value errors less accurately than Memcheck.

One interesting group of DBA tools are those that use shadow values. These tools shadow, purely in software, every register and memory value with another value that says something about it. We call these shadow value tools.

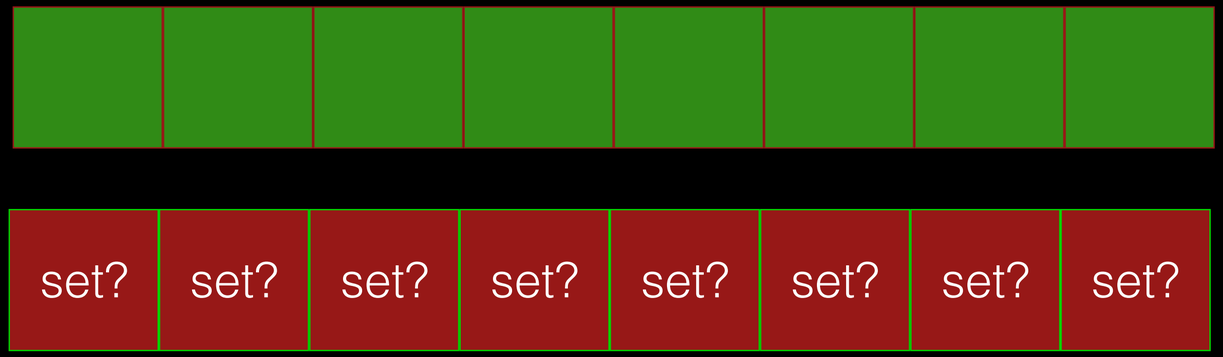
0xc000000



```
95  /*------------------------------------------------------------*/
96  /*--- V bits and A bits                                    ---*/
97  /*------------------------------------------------------------*/
98
99  /* Conceptually, every byte value has 8 V bits, which track whether Memcheck
100    thinks the corresponding value bit is defined.  And every memory byte
101    has an A bit, which tracks whether Memcheck thinks the program can access
102    it safely (ie. it's mapped, and has at least one of the RWX permission bits
103    set).  So every N-bit register is shadowed with N V bits, and every memory
104    byte is shadowed with 8 V bits and one A bit.
105
106    In the implementation, we use two forms of compression (compressed V bits
107    and distinguished secondary maps) to avoid the 9-bit-per-byte overhead
108    for memory.
109
110    Memcheck also tracks extra information about each heap block that is
111    allocated, for detecting memory leaks and other purposes.
112 */
```

memcheck/mc_main.c

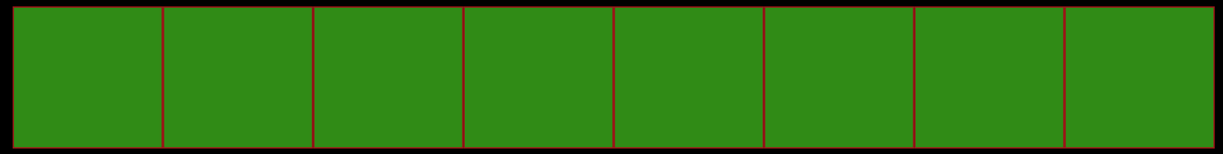
0xc000000




```
95  /*--------------------------------------------------------------*/
96  /*--- V bits and A bits                                      ---*/
97  /*--------------------------------------------------------------*/
98
99  /* Conceptually, every byte value has 8 V bits, which track whether Memcheck
100    thinks the corresponding value bit is defined.  And every memory byte
101    has an A bit, which tracks whether Memcheck thinks the program can access
102    it safely (ie. it's mapped, and has at least one of the RWX permission bits
103    set).  So every N-bit register is shadowed with N V bits, and every memory
104    byte is shadowed with 8 V bits and one A bit.
105
106    In the implementation, we use two forms of compression (compressed V bits
107    and distinguished secondary maps) to avoid the 9-bit-per-byte overhead
108    for memory.
109
110    Memcheck also tracks extra information about each heap block that is
111    allocated, for detecting memory leaks and other purposes.
112 */
```

memcheck/mc_main.c

0xc000000

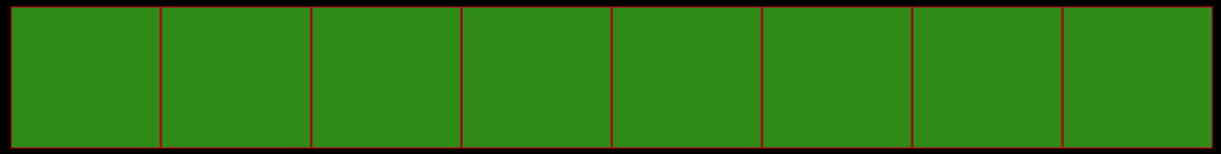


valid?	set?	set?	set?	set?	set?	set?	set?	set?

```
95  /*------------------------------------------------------------*/
96  /*--- V bits and A bits                                    ---*/
97  /*------------------------------------------------------------*/
98
99  /* Conceptually, every byte value has 8 V bits, which track whether Memcheck
100    thinks the corresponding value bit is defined.  And every memory byte
101    has an A bit, which tracks whether Memcheck thinks the program can access
102    it safely (ie. it's mapped, and has at least one of the RWX permission bits
103    set).  So every N-bit register is shadowed with N V bits, and every memory
104    byte is shadowed with 8 V bits and one A bit.
105
106    In the implementation, we use two forms of compression (compressed V bits
107    and distinguished secondary maps) to avoid the 9-bit-per-byte overhead
108    for memory.
109
110    Memcheck also tracks extra information about each heap block that is
111    allocated, for detecting memory leaks and other purposes.
112 */
```

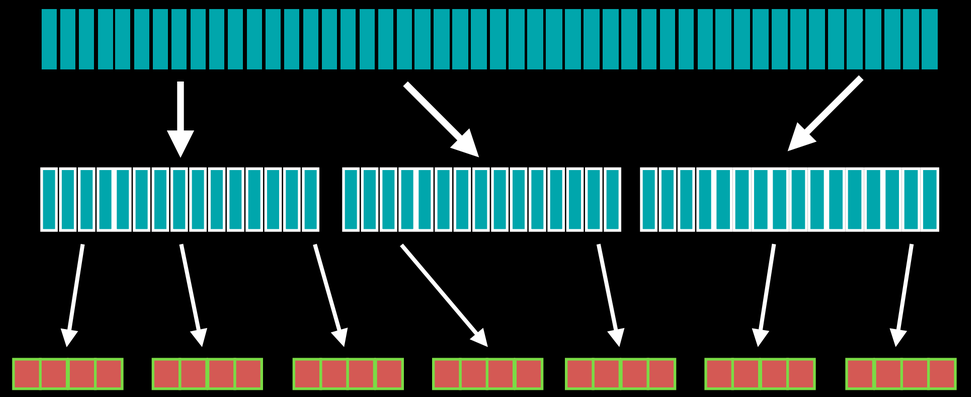
memcheck/mc_main.c

0xc000000



valid?	set?	set?	set?	set?	set?	set?	set?	set?

```
107     and distinguished secondary maps) to avoid the 9-bit-per-byte overhead
108     for memory.
109
110     Memcheck also tracks extra information about each heap block that is
111     allocated, for detecting memory leaks and other purposes.
112  */
113
114  /*------------------------------------------------------------*/
115  /*--- Basic A/V bitmap representation.                     ---*/
116  /*------------------------------------------------------------*/
117
118  /* All reads and writes are checked against a memory map (a.k.a. shadow
119     memory), which records the state of all memory in the process.
120
121     On 32-bit machines the memory map is organised as follows.
122     The top 16 bits of an address are used to index into a top-level
123     map table, containing 65536 entries.  Each entry is a pointer to a
124     second-level map, which records the accesibililty and validity
125     permissions for the 65536 bytes indexed by the lower 16 bits of the
126     address.  Each byte is represented by two bits (details are below).  So
127     each second-level map contains 16384 bytes.  This two-level arrangement
128     conveniently divides the 4G address space into 64k lumps, each size 64k
129     bytes.
```



memcheck/mc_main.c

Nethercote et al. **Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation**

PLDI '07

Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation

Nicholas Nethercote
National ICT Australia, Melbourne, Australia
njn@csse.unimelb.edu.au

Julian Seward
OpenWorks LLP, Cambridge, UK
julian@open-works.co.uk

Abstract

Dynamic binary instrumentation (DBI) frameworks make it easy to build dynamic binary analysis (DBA) tools such as checkers and profilers. Much of the focus on DBI frameworks has been on performance; little attention has been paid to their capabilities. As a result, we believe the potential of DBI has not been fully exploited.

In this paper we describe Valgrind, a DBI framework designed for building heavyweight DBA tools. We focus on its unique support for *shadow values*—a powerful but previously little-studied and difficult-to-implement DBA technique, which requires a tool to shadow every register and memory value with another value that describes it. This support accounts for several crucial design features that distinguish Valgrind from other DBI frameworks. Because of these features, lightweight tools built with Valgrind run comparatively slowly, but Valgrind can be used to build more interesting, heavyweight tools that are difficult or impossible to build with other DBI frameworks such as Pin and DynamoRIO.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—debugging aids, monitors; D.3.4 [*Programming Languages*]: Processors—incremental compilers

General Terms Design, Performance, Experimentation

Keywords Valgrind, Memcheck, dynamic binary instrumentation, dynamic binary analysis, shadow values

1. Introduction

Valgrind is a dynamic binary instrumentation (DBI) framework that occupies a unique part of the DBI framework design space. This paper describes how it works, and how it differs from other frameworks.

1.1 Dynamic Binary Analysis and Instrumentation

Many programmers use program analysis tools, such as error checkers and profilers, to improve the quality of their software. *Dynamic binary analysis* (DBA) tools are one such class of tools; they analyse programs at run-time at the level of machine code.

DBA tools are often implemented using *dynamic binary instrumentation* (DBI), whereby the *analysis code* is added to the original code of the *client program* at run-time. This is convenient for users, as no preparation (such as recompiling or relinking) is needed. Also, it gives 100% instrumentation coverage of user-mode code, without requiring source code. Several generic *DBI frameworks* exist, such as Pin [11], DynamoRIO [3], and Valgrind [18, 15]. They provide a base system that can instrument and run code, plus an environment for writing tools that plug into the base system.

The performance of DBI frameworks has been studied closely [1, 2, 9]. Less attention has been paid to their instrumentation capabilities, and the tools built with them. This is a shame, as it is the tools that make DBI frameworks useful, and complex tools are more interesting than simple tools. As a result, we believe the potential of DBI has not been fully exploited.

1.2 Shadow Value Tools and Heavyweight DBA

One interesting group of DBA tools are those that use *shadow values*. These tools shadow, purely in software, every register and memory value with another value that says something about it. We call these *shadow value tools*. Consider the following motivating list of shadow value tools; the descriptions are brief but demonstrate that shadow values (a) can be used in a wide variety of ways, and (b) are powerful and interesting.

Memcheck [25] uses shadow values to track which bit values are undefined (i.e. uninitialised, or derived from undefined values) and can thus detect dangerous uses of undefined values. It is used by thousands of C and C++ programmers, and is probably the most widely-used DBA tool in existence.[1]

TaintCheck [20] tracks which byte values are tainted (i.e. from an untrusted source, or derived from tainted values) and can thus detect dangerous uses of tainted values. *TaintTrace* [6] and *LIFT* [23] are similar tools.

McCamant and Ernst's secret-tracking tool [13] tracks which bit values are secret (e.g. passwords), and determines how much information about secret inputs is revealed by public outputs.

Hobbes [4] tracks each value's type (determined from operations performed on the value) and can thus detect subsequent operations inappropriate for a value of that type.

DynCompB [7] similarly determines abstract types of byte values, for program comprehension and invariant detection purposes.

Annelid [16] tracks which word values are array pointers, and from this can detect bounds errors.

Redux [17] creates a *dynamic dataflow graph*, a visualisation of a program's entire computation; from the graph one can see all the prior operations that contributed to the each value's creation.

In these tools each shadow value records a simple approximation of each value's history—e.g. one shadow bit per bit, one

[1] Purify [8] is a memory-checking tool similar to Memcheck. However, Purify is not a shadow value tool as it does not does not track definedness of values through registers. As a result, it detects undefined value errors less accurately than Memcheck.

PLDI'07 June 11–13, 2007, San Diego, California, USA.

Nethercote et al. **Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation**

PLDI '07

the core passes to it. Writing a new tool plug-in (and thus a new Valgrind tool) is much easier than writing a new DBA tool from

Most DBI frameworks use injection-style methods rather than having their own program loader. As well as avoiding the problems encountered by the prior two approaches, our third approach has two other advantages. First, it gives Valgrind great control over memory layout. Second, it it avoids dependencies on other tools such as the dynamic linker, which we have found to be an excellent strategy for improving robustness.[3]

3.4 Guest and Host Registers

Valgrind itself runs on the machine's real or *host* CPU, and (conceptually) runs the client program on a simulated or *guest* CPU. We refer to registers in the host CPU as *host registers* and those of the simulated CPU as *guest registers*. Due to the dynamic binary recompilation process, a guest register's value may reside in one of the host's registers, or it may be spilled to memory for a variety of reasons. Shadow registers are shadows of guest registers.

Valgrind provides a block of memory per client thread called the ThreadState. Each one contains space for all the thread's guest and shadow registers and is used to hold them at various times, in particular between each code block. Storing guest registers in memory between code blocks sounds like a bad idea at first, because it means that they must be moved between memory and the host registers frequently, but it is reasonable for heavyweight tools with high host register pressure for which the benefits of a more optimistic strategy are greatly diminished.

3.5 Representation of code: D&R vs. C&A

There are two fundamental ways for a DBI framework to represent code and allow instrumentation.

Valgrind uses *disassemble-and-resynthesise (D&R)*: machine code is converted to an IR in which each instruction becomes one or more IR operations. This IR is instrumented (by adding more IR) and then converted back to machine code. All of the original code's effects on guest state (e.g. condition code setting) must be explicitly represented in the IR because the original client instructions are discarded and the final code is generated purely from the IR. Valgrind's use of D&R is the single feature that most distinguishes it from other DBI frameworks.

Other DBI frameworks use *copy-and-annotate (C&A)*: incoming instructions are copied through verbatim except for necessary control flow changes. Each instruction is annotated with a description of its effects, via data structures (e.g. DynamoRIO) or an instruction-querying API (e.g. Pin). Tools use the annotations to guide their instrumentation. The added analysis code must must be interleaved with the original code without perturbing its effects.

Hybrid approaches are possible. For example, earlier versions of Valgrind used D&R for integer instructions and C&A for FP and SIMD instructions (this was more by accident than design). Variations are also possible; for example, DynamoRIO allows instruction bytes to be modified in-place before being copied through.

Each approach has its pros and cons, depending on the instrumentation requirements. D&R may require more up-front design and implementation effort, because a D&R representation is arguably more complex. Also, generating good code at the end requires more development effort—Valgrind's JIT uses a lot of conventional compiler technology. In contrast, for C&A, good client code stays good with less effort. A D&R JIT compiler will probably also translate code more slowly.

D&R may not suitable for some tools that require low-level information. For example, the exact opcode used by each instruc-

[3] For example, Valgrind no longer uses the standard C library, but has a small version of its own. This has avoided any potential complications caused by having two copies of the C library in the address space—one for the client, and and for Valgrind and the tool. It also made the AIX port much easier, because AIX's C library is substantially different to Linux's.

memory mappings to force components into the right place, which turned out to be somewhat unreliable.

tion may be lost. IR annotations can help, however—for example, Valgrind has "marker" statements that indicate the boundaries, addresses and lengths of original instructions. C&A can suffer the same problem if the annotations are not comprehensive.

D&R's strengths emerge when complex analysis code must be added. First, D&R's use of the same IR for both client and analysis code guarantees that analysis code is as expressive and powerful as client code. Making all side-effects explicit (e.g. condition code computations) can make instrumentation easier.

The performance dynamics also change. The JIT compiler can optimise analysis code and client code equally well, and naturally tightly interleaves the two. In contrast, C&A must provide a separate way to describe analysis code (so C&A requires some kind of IR after all). This code must then be fitted around the original instructions, which requires effort (either by the framework or the tool-writer) to do safely and with good performance. For example, Pin analysis code is written as C functions (i.e. the analysis code IR is C), which are compiled with an external C compiler, and Pin then inlines them if possible, or inserts calls to them.

Finally, D&R is more verifiable—any error converting machine code to IR is likely to cause visibly wrong behaviour, whereas a C&A annotation error will result in incorrect analysis of a correctly behaving client.[4] D&R also permits binary translation from one platform to another (although Valgrind does not do this). D&R also allows the original code's behaviour to be arbitrarily changed.

In summary, D&R requires more effort up-front and is overkill for lightweight instrumentation. However, it naturally supports heavyweight instrumentation such as that required by shadow value tools, and so is a natural fit for Valgrind.

3.6 Valgrind's IR

Prior to version 3.0.0 (August 2005), Valgrind had an x86-specific, part D&R, part C&A, assembly-code-like IR in which the units of translation were basic blocks. Since then Valgrind has had an architecture-neutral, D&R, single-static-assignment (SSA) IR that is more similar to what might be used in a compiler. IR blocks are *superblocks*: single-entry, multiple-exit stretches of code.

Each IR block contains a list of statements, which are operations with side-effects, such as register writes, memory stores, and assignments to temporaries. Statements contain expressions, which represent pure (no side effects) values such as constants, register reads, memory loads, and arithmetic operations. For example, a store statement contains one expression for the store address and another for the store value. Expressions can be arbitrarily complicated trees (*tree IR*), but they can also be flattened by introducing statements that write intermediate values to temporaries (*flat IR*).

The IR has some RISC-like features: it is load/store, each primitive operation only does one thing (many CISC instructions are broken up into multiple operations), and when flattened, all operations operate only on temporaries and literals. Nonetheless, supporting all the standard integer, FP and SIMD operations of different sizes requires more than 200 primitive arithmetic/logical operations.

The IR is architecture-independent. Valgrind handles unusual architecture-specific instructions, such as `cpuid` on x86, with a call to a C function that emulates the instruction. These calls have annotations that say which guest registers and memory locations they access, so that a tool can see some of their effects while avoiding the need for Valgrind to represent the instruction explicitly in the IR. This is another case (like the "marker" statements) where Valgrind uses IR annotations to facilitate instrumentation (but it is not C&A, because the instruction is emulated, not copied through).

[4] This is not just a theoretical concern. Valgrind's old IR used C&A for SIMD instructions; some SIMD loads were mis-annotated as stores, and some SIMD stores as loads, for more than a year before being noticed.

copy-and-annotate: incoming instructions are copied through verbatim except for necessary control flow changes. Tools use the annotations to guide their instrumentation.

disassemble-and-resynthesise: machine code is converted to an IR in which each instruction becomes one or more IR operations. This IR is instrumented and then converted back to machine code.

Nethercote et al. **Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation**

PLDI '07

for the client, and and for Valgrind and the tool. It also made the AIX port much easier, because AIX's C library is substantially different to Linux's.

5.3 Tool Instrumentation Capabilities

In this section, we compare Valgrind's support for all nine shadow value requirements against Pin [11], because Pin is the best known of the currently available DBI frameworks, and the one that has the most support for shadow values (after Valgrind). The following comparison is based on discussions with two Pin developers [10].

Pin supports R5 (instrument start-up allocations), R8 (instrument heap (de)allocations) and R9 (extra output) directly. It does not support R6 (instrument system call (de)allocations) and R7 (instrument stack (de)allocations) directly, but provides features that allow a Pin tool to manually support them fairly easily.

For R1 (provide shadow registers) Pin provides "virtual registers" which are register-allocated along with guest registers and saved in memory when a thread is not running. Shadow registers could be stored in them. However, virtual registers are not fully first-class citizens. For example, there are no 128-bit virtual registers, so 128-bit SIMD registers cannot be fully shadowed, which would prevent some tools (e.g. Memcheck) from working fully.

Pin provides no built-in support for R2 (provide shadow memory), so tools must cope with the non-atomicity of loads/stores and shadow loads/stores themselves.[10] For example, the Pin tool called *pinSEL* [14], which uses shadow memory but not full shadow values, sets and checks an extra *interference bit* on every shadow load. This lets it handle any thread switches or asynchronous signals that occur between a load/store and a shadow load/store (both of which can occur even on uni-processors under Pin). Multi-threaded programs running on multi-processors are even trickier, and pinSEL does not handle them. In comparison, Valgrind's thread serialisation and asynchronous signal treatment frees shadow value tools from having to deal with this issue.

For R3 (instrument read/write instructions) Pin allows all register and memory accesses to be seen. However, analysis code in Pin is written as C functions, which can be inlined if they contain no control flow. This means that SIMD instructions are again a problem; if a tool needs to use SIMD instructions in its analysis code (as Memcheck does), these would have to be written in Pin using (platform-specific) inline assembly code. This is caused by Pin using C&A and its method for writing analysis code (C code) having less expressivity than client code (machine code).

R4 (instrument read/write system calls) is another stumbling block; it can be done manually within a tool via Pin's system call instrumentation, but would require a large effort—each shadow value tool would essentially need to reimplement Valgrind's system call wrappers.

5.4 Tool Performance

We performed experiments on 25 of the 26 SPEC CPU2000 benchmarks (we could not run `galgel` as gfortran failed to compile it). We ran them with the "reference" inputs in 32-bit mode on a 2.4 GHz Intel Core 2 Duo with 1GB RAM and a 4MB L2 cache running SUSE Linux 10.2, kernel 2.6.18.2. We compared several tools built with Valgrind 3.2.1: (a) Nulgrind, the "no instrumentation" tool; (b) ICntI, an instruction counter which uses inline code to increment a counter for every instruction executed; (c) ICntC, like ICntI but uses a C function call to increment the counter; and (d) Memcheck (with leak-checking off, because it runs at program termination and so would cloud the comparison). Table 2 shows the slow-down factors of these tools.

Lightweight tools. The mean slow-down of 4.3x for the no-instrumentation case (Nulgrind) is high compared to other frameworks. This is consistent with other researchers' findings—a previous comparison [11] showed that Valgrind is 4.0x slower than Pin and 4.4x slower than DynamoRIO on the SPEC CPU2000 integer benchmarks in the no-instrumentation case, and 3.3x and 2.0x slower for a lightweight basic block counting tool.[11]

Re-implementing chaining in Valgrind would improve these cases somewhat. However, these lightweight tools are exactly the kinds of tools that Valgrind is *not* targeted at, and Valgrind will never be as fast as Pin or DynamoRIO for these cases. For example, consider Valgrind's use of a D&R representation. For a simple tool like a basic block counter, D&R makes no sense. Rather, the use of D&R is targeted towards heavyweight tools. For this reason, we do not repeat such comparisons with lightweight tools.

The difference between ICntI and ICntC shows the advantage of inline code over C calls. ICntI could be further improved by batching counter increments together.

Heavyweight tools built with Valgrind. Memcheck's mean slow-down factor is 22.2x. Other shadow value tools built with Valgrind have similar or worse slow-downs. TaintCheck ran 37x slower on an invocation of bzip2 [20], but had better performance on an I/O-bound invocation of the Apache web server. Annelid ran a subset of the SPEC CPU2000 benchmarks ("train" inputs) 35.2x slower than native [16]. McCamant and Ernst's secret tracker has slow-downs "similar to Memcheck... 10–100x for CPU-bound programs" [13]. Redux did much more expensive analysis and was not practical for anything more than toy programs [17]. Slow-down figures are not available for DynCompB [7].

SIMD instructions; some SIMD loads were mis-annotated as stores, and some SIMD stores as loads, for more than a year before being noticed.

Program	Nat. (s)	Nulg.	ICntI	ICntC	Memc.
bzip2	192.7	3.5	7.2	10.5	16.1
crafty	92.4	6.9	12.3	22.5	36.0
eon	408.5	7.5	11.8	21.0	51.4
gap	131.3	4.0	9.1	13.5	25.5
gcc	90.0	5.3	9.0	14.1	39.0
gzip	212.1	3.2	5.9	9.0	14.7
mcf	87.0	2.0	3.5	5.4	7.0
parser	218.9	3.6	7.0	10.4	17.8
perlbmk	179.6	4.8	9.6	14.6	27.1
twolf	262.5	3.1	6.5	10.7	16.0
vortex	86.7	6.5	11.4	17.8	38.7
vpr	149.4	4.1	7.7	11.3	16.4
ammp	345.2	3.4	6.5	9.1	32.7
applu	583.0	5.2	14.1	28.1	19.7
apsi	469.0	3.4	8.2	12.5	16.4
art	100.4	4.7	9.4	13.7	24.0
equake	118.2	3.8	8.4	12.4	17.1
facerec	280.9	4.7	8.2	12.2	17.4
fma3d	284.7	4.1	9.4	16.2	26.0
lucas	183.5	3.7	7.1	10.8	24.8
mesa	148.9	5.9	10.3	15.9	57.9
mgrid	809.1	3.5	9.8	14.4	16.9
sixtrack	355.7	5.6	13.4	18.3	20.2
swim	388.2	3.2	11.9	15.3	10.7
wupwise	192.1	7.4	11.8	17.3	26.7
geo. mean		4.3	8.8	13.5	22.1

Table 2. Performance of four Valgrind tools on SPEC CPU2000. Column 1 gives the program name; integer programs are listed before floating-point programs. Column 2 gives the native execution time in seconds. Columns 3–6 give the slow-down factors for each tool. The final row shows each column's geometric mean.

[10] It does have thread-locking primitives, but they would be too coarse-grained to be practical for use with shadow memory.

[11] But the measured Valgrind tool used a C function to increment the counter; the use of inline code would have narrowed the gap.

"Pin provides no built-in support for [shadow memory], so tools must cope with the non-atomicity of loads/stores and shadow loads/stores themselves."

"Valgrind's thread serialisation and asynchronous signal treatment frees shadow value tools from having to deal with this issue."

Luk et al. **Pin: Building Customized Program Analysis Tools w/ Dynamic Instrumentation**

PLDI ‘05

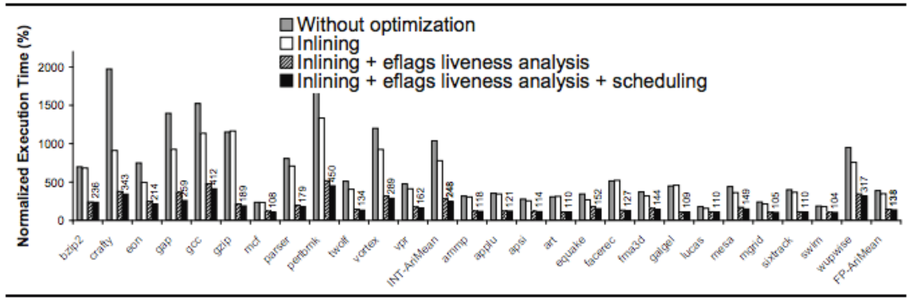


Figure 6. Performance of Pin with basic-block counting instrumentation on the IA32 architecture.

analysis, reducing the average slowdown to 2.8x for integer and 1.5x for floating point. Scheduling of instrumentation code further reduces the slowdown to 2.5x for integer and 1.4x for floating point.

4.3 Performance Comparison with Valgrind and DynamoRIO

We now compare the performance of Pin against Valgrind and DynamoRIO. Valgrind is a popular instrumentation tool on Linux and is the only binary-level JIT other than Pin that re-allocates registers. DynamoRIO is generally regarded as the performance leader in binary-level dynamic optimization. We used the latest release of each tool for this experiment: Valgrind 2.2.0 [22] and DynamoRIO 0.9.3 [6]. We ran two sets of experiments: one without instrumentation and one with basic-block counting instrumentation. We implemented basic-block counting by modifying a tool in the Valgrind package named `lackey` and a tool in the DynamoRIO package named `countcalls`. We show only the integer results in Figure 7 as integer codes are more problematic than floating-point codes in terms of the slowdown caused by instrumentation.

Figure 7(a) shows that without instrumentation both Pin and DynamoRIO significantly outperform Valgrind. DynamoRIO is faster than Pin on `gcc`, `perlbmk` and `vortex`, mainly because Pin spends more jitting time in these three benchmarks (refer back to Figure 5(a) for the breakdown) than DynamoRIO, which does not re-allocate registers. Pin is faster than DynamoRIO on a few benchmarks such as `crafty` and `gap` possibly because of the advantages that Pin has in indirect linking (i.e. incremental linking, cloning, and local hash tables). Overall, DynamoRIO is 12% faster than Pin without instrumentation. Given that DynamoRIO was primarily designed for *optimization*, the fact that Pin can come this close is quite acceptable.

When we consider the performance with instrumentation shown in Figure 7(b), Pin outperforms both DynamoRIO and Valgrind by a significant margin: on average, Valgrind slows the application down by 8.3 times, DynamoRIO by 5.1 times, and Pin by 2.5 times. Valgrind inserts a call before every basic block's entry but it does not automatically inline the call. For DynamoRIO, we use its low-level API to update the counter inline. Nevertheless, DynamoRIO still has to save and restore the `eflags` explicitly around each counter update. In contrast, Pin automatically inlines the call and performs liveness analysis to eliminate unnecessary `eflags` save/restore. This clearly demonstrates a main advantage of Pin: it provides efficient instrumentation without shifting the burden to the Pintool writer.

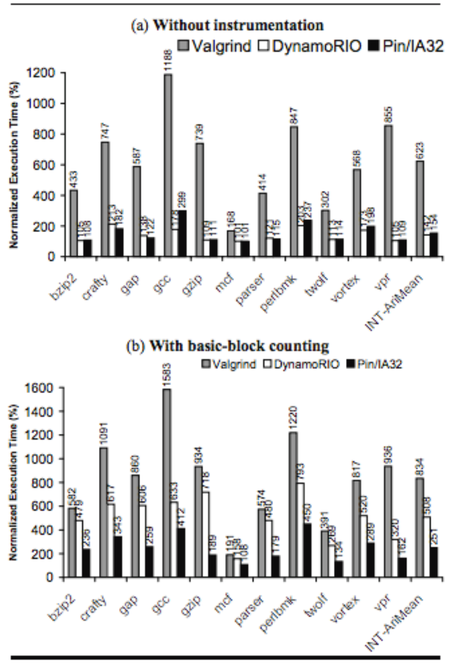


Figure 7. Performance comparison among Valgrind, DynamoRIO, and Pin. `Eon` is excluded because DynamoRIO does not work on the icc-generated binary of this benchmark. Omitting `eon` causes the two arithmetic means of Pin/IA32 slightly different than the ones shown in Figures 5(a) and 6.

“in Figure 7(b), Pin outperforms both DynamoRIO and Valgrind by a significant margin: on average, Valgrind slows the application down by 8.3 times [...] and Pin by 2.5 times.”

- Pin: a “copy and annotate” system
  - \+ Lower overhead than the alternatives
  - \+ Close-to-the-metal instrumentation API
  - \- Best for tracking control flow, rather than memory
- Valgrind: a “disassemble and resynthsize” system
  - \+ Richer API and shadow memory allows for per-memory word metadata tracking
  - \- Greater runtime overhead

State tracking in practice

problem statement

“What values did a piece of memory have over time?”

“How does a value propagate through a running system?”

Chow et al. **Understanding Data Lifetime via Whole System Simulation**

USENIX Sec '04

Understanding Data Lifetime via Whole System Simulation

Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, Mendel Rosenblum
{jchow,blp,talg,kchristo,mendel}@cs.stanford.edu
Stanford University Department of Computer Science

Abstract

Strictly limiting the lifetime (i.e. propagation and duration of exposure) of sensitive data (e.g. passwords) is an important and well accepted practice in secure software development. Unfortunately, there are no current methods available for easily analyzing data lifetime, and very little information available on the quality of today's software with respect to data lifetime.

We describe a system we have developed for analyzing sensitive data lifetime through whole system simulation called TaintBochs. TaintBochs tracks sensitive data by "tainting" it at the hardware level. Tainting information is then propagated across operating system, language, and application boundaries, permitting analysis of sensitive data handling at a whole system level.

We have used TaintBochs to analyze sensitive data handling in several large, real world applications. Among these were Mozilla, Apache, and Perl, which are used to process millions of passwords, credit card numbers, etc. on a daily basis. Our investigation reveals that these applications and the components they rely upon take virtually no measures to limit the lifetime of sensitive data they handle, leaving passwords and other sensitive data scattered throughout user and kernel memory. We show how a few simple and practical changes can greatly reduce sensitive data lifetime in these applications.

1 Introduction

Examining sensitive data lifetime can lend valuable insight into the security of software systems. When studying data lifetime we are concerned with two primary issues: how long a software component (e.g. operating system, library, application) keeps data it is processing alive (i.e. in an accessible form in memory or persistent storage) and where components propagate data (e.g. buffers, log files, other components).

As data lifetime increases so does the likelihood of exposure to an attacker. Exposure can occur by way of an attacker gaining access to system memory or to persistent storage (e.g. swap space) to which data has leaked. Careless data handling also increases the risk of data exposure via interaction with features such as logging, command histories, session management, crash dumps or crash reporting [6], interactive error reporting, etc.

Unfortunately, even simple questions about data lifetime can be surprisingly difficult to answer in real systems. The same data is often handled by many different components, including device drivers, operating system, system libraries, programming language runtimes, applications, etc., in the course of a single transaction. This limits the applicability of traditional static and dynamic program analysis techniques, as they are typically limited in scope to a single program, often require program source code, and generally cannot deal with more than one implementation language.

To overcome these limitations we have developed a tool based on whole-system simulation called TaintBochs, which allows us to track the propagation of sensitive data at hardware level, enabling us to examine *all* places that sensitive data can reside. We examine systems with TaintBochs by running the entire software stack, including operating system, application code, etc. inside a simulated environment. Every byte of system memory, device state, and relevant processor state is tagged with a taint-status flag. Data is "tainted" if it is considered sensitive.

TaintBochs propagates taint flags whenever their corresponding values in hardware are involved in an operation. Thus, tainted data is tracked throughout the system as it flows through kernel device drivers, user-level GUI widgets, application buffers, etc. Tainting is introduced when sensitive data enters the system, such as when a password is read from the keyboard device, an application reads a particular data set, etc.

We applied TaintBochs to analyzing the lifetime of password information in a variety of large, real-world applications, including Mozilla, Apache, Perl, and Emacs on the Linux platform. Our analysis revealed that these applications, the kernel, and the libraries that they relied upon generally took no steps to reduce data lifetime. Buffers containing sensitive data were deallocated without being cleared of their contents, leaving sensitive data to sit on the heap indefinitely. Sensitive data was left in cleartext in memory for indeterminate periods without good reason, and unnecessary replication caused excessive copies of password material to be scattered all over the heap. In the case of Emacs our analysis also uncovered an interaction between the keyboard history

mits incredible control, allowing us to augment the ar-

If any bit in a byte is tainted, the entire byte is consid-

TaintBochs tracks sensitive data by "tainting" it at the hardware level. Tainting information is then propagated across operating system, language, and application boundaries, permitting analysis of sensitive data handling at a whole system level.

Chow et al. **Understanding Data Lifetime via Whole System Simulation** USENIX Sec '04

Understanding Data Lifetime via Whole System Simulation

Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, Mendel Rosenblum
{jchow,blp,talg,kchristo,mendel}@cs.stanford.edu
Stanford University Department of Computer Science

Abstract

Strictly limiting the lifetime (i.e. propagation and duration of exposure) of sensitive data (e.g. passwords) is an important and well accepted practice in secure software development. Unfortunately, there are no current methods available for easily analyzing data lifetime, and very little information available on the quality of today's software with respect to data lifetime.

We describe a system we have developed for analyzing sensitive data lifetime through whole system simulation called TaintBochs. TaintBochs tracks sensitive data by "tainting" it at the hardware level. Tainting information is then propagated across operating system, language, and application boundaries, permitting analysis of sensitive data handling at a whole system level.

We have used TaintBochs to analyze sensitive data handling in several large, real world applications. Among these were Mozilla, Apache, and Perl, which are used to process millions of passwords, credit card numbers, etc. on a daily basis. Our investigation reveals that these applications and the components they rely upon take virtually no measures to limit the lifetime of sensitive data they handle, leaving passwords and other sensitive data scattered throughout user and kernel memory. We show how a few simple and practical changes can greatly reduce sensitive data lifetime in these applications.

1 Introduction

Examining sensitive data lifetime can lend valuable insight into the security of software systems. When studying data lifetime we are concerned with two primary issues: how long a software component (e.g. operating system, library, application) keeps data it is processing alive (i.e. in an accessible form in memory or persistent storage) and where components propagate data (e.g. buffers, log files, other components).

As data lifetime increases so does the likelihood of exposure to an attacker. Exposure can occur by way of an attacker gaining access to system memory or to persistent storage (e.g. swap space) to which data has leaked. Careless data handling also increases the risk of data exposure via interaction with features such as logging, command histories, session management, crash dumps or crash reporting [6], interactive error reporting, etc.

Unfortunately, even simple questions about data lifetime can be surprisingly difficult to answer in real systems. The same data is often handled by many different components, including device drivers, operating system, system libraries, programming language runtimes, applications, etc., in the course of a single transaction. This limits the applicability of traditional static and dynamic program analysis techniques, as they are typically limited in scope to a single program, often require program source code, and generally cannot deal with more than one implementation language.

To overcome these limitations we have developed a tool based on whole-system simulation called TaintBochs, which allows us to track the propagation of sensitive data at hardware level, enabling us to examine *all* places that sensitive data can reside. We examine systems with TaintBochs by running the entire software stack, including operating system, application code, etc. inside a simulated environment. Every byte of system memory, device state, and relevant processor state is tagged with a taint-status flag. Data is "tainted" if it is considered sensitive.

TaintBochs propagates taint flags whenever their corresponding values in hardware are involved in an operation. Thus, tainted data is tracked throughout the system as it flows through kernel device drivers, user-level GUI widgets, application buffers, etc. Tainting is introduced when sensitive data enters the system, such as when a password is read from the keyboard device, an application reads a particular data set, etc.

We applied TaintBochs to analyzing the lifetime of password information in a variety of large, real-world applications, including Mozilla, Apache, Perl, and Emacs on the Linux platform. Our analysis revealed that these applications, the kernel, and the libraries that they relied upon generally took no steps to reduce data lifetime. Buffers containing sensitive data were deallocated without being cleared of their contents, leaving sensitive data to sit on the heap indefinitely. Sensitive data was left in cleartext in memory for indeterminate periods without good reason, and unnecessary replication caused excessive copies of password material to be scattered all over the heap. In the case of Emacs our analysis also uncovered an interaction between the keyboard history

mits incredible control, allowing us to augment the ar-

If any bit in a byte is tainted, the entire byte is consid-

We have used TaintBochs to analyze sensitive data handling in several large, real world applications. Among these were Mozilla, Apache, and Perl, which are used to process millions of passwords, credit card numbers, etc. on a daily basis.

Chow et al. **Understanding Data Lifetime via Whole System Simulation** USENIX Sec '04

tecture with taint propagation, extend the instruction set, etc.

We have augmented Bochs with three capabilities to produce TaintBochs. First, we provide the ability to track the propagation of sensitive data through the system at a hardware level, i.e. tainting. Second, we have added logging capabilities that allow system state such as memory and registers at any given time during a system's execution history to be examined. Finally, we developed an analysis framework that allows information about OS internals, debug information for the software that is running, etc. to be utilized in an integrated fashion to allow easy interpretation of tainting information. This allows us to trace tainted data to an exact program variable in an application (or the kernel) in the guest, and code propagating tainting to an exact source file and line number.

Our basic usage model consists of two phases. First, we run a simulation in which sensitive data (e.g. coming from the keyboard, network, etc.) is identified as tainted. The workload consists of normal user interaction, e.g. logging into a website via a browser. In the second phase, the simulation data is analyzed with the analysis framework. This allows us to answer open-ended queries about the simulation, such as where tainted data came from, where it was stored, how it was propagated, etc.

We will begin by looking at the implementation of TaintBochs, focusing on modifications to the simulator to facilitate tainting, logging, etc. We will then move on to examine the analysis framework and how it can be used with other tools to gain a complete picture of data lifetime in a system.

3.1 Hardware Level Tainting

There are two central issues to implementing hardware level tainting: first, tracking the location of sensitive state in the system, and, second, deciding how to evolve that state over time to keep a consistent picture of which state is sensitive. We will examine each of these issues in turn.

Shadow Memory To track the location of sensitive data in TaintBochs, we added another memory, set of registers, etc. called a *shadow memory*. The shadow memory tracks taint status of every byte in the system. Every operation performed on machine state by the processor or devices causes a parallel operation to be performed in shadow memory, e.g. copying a word from register A to location B causes the state in the shadow register A to be copied to shadow location B. Thus to determine if a byte is tainted we need only look in the corresponding location in shadow memory.

cted tainted. Maintaining taint status at a byte granularity is a conservative approximation, i.e. we do not ever lose track of sensitive data, although some data may be unnecessarily tainted. Bit granularity would take minimal additional effort, but we have not yet encountered a situation where this would noticeably aid our analysis.

For simplicity, TaintBochs only maintains shadow memory for the guest's main memory and the IA-32's eight general-purpose registers. Debug registers, control registers, SIMD (e.g. MMX, SSE) registers, and flags are disregarded, as is chip set and I/O device state. Adding the necessary tracking for other processor or I/O device state (e.g. disk, frame buffer) would be quite straightforward, but the current implementation is sufficient for many kinds of useful analysis. We are not terribly concerned about the guest's ability to launder taint bits through the processor's debug registers, for example, as our assumption is that software under analysis is not intentionally malicious.

Propagation Policy We must decide how operations in the system should affect shadow state. If two registers A and B are added, and one of them is tainted, is the register where the result are stored also tainted? We refer to the collective set of policies that decide this as the *propagation policy*.

In the trivial case where data is simply copied, we perform the same operation in the address space of shadow memory. So, if the assignment $A \leftarrow B$ executes on normal memory, then $A \leftarrow B$ is also executed on shadow memory. Consequently, if B was tainted then A is now also tainted, and if B was not tainted, A is now also no longer tainted.

The answer is less straightforward when an instruction produces a new value based on a set of inputs. In such cases, our simulator must decide on whether and how to taint the instruction's output(s). Our choices must balance the desire to preserve any possibly interesting taints against the need to minimize spurious reports, i.e. avoid tainting too much data or uninteresting data. This roughly corresponds to the false negatives vs. false positives trade-offs made in other taint analysis tools. As we will see, it is in general impossible to achieve the latter goal perfectly, so some compromises must be made.

Processor instructions typically produce outputs that are some function of their inputs. Our basic propagation policy is simply that *if any byte of any input value is tainted, then all bytes of the output are tainted.* This policy is clearly *conservative* and errs on the side of tainting too much. Interestingly though, with the exception of cases noted below, we have not yet encountered any obviously spurious output resulting from our policy.

"We must decide how operations in the system should affect shadow state. If two registers A and B are added, and one of them is tainted, is the register where the result are stored also tainted? We refer to the collective set of policies that decide this as the propagation policy."

Chow et al. **Understanding Data Lifetime via Whole System Simulation** USENIX Sec '04

chitecture with taint propagation, extend the instruction set, etc.

We have augmented Bochs with three capabilities to produce TaintBochs. First, we provide the ability to track the propagation of sensitive data through the system at a hardware level, i.e. tainting. Second, we have added logging capabilities that allow system state such as memory and registers at any given time during a system's execution history to be examined. Finally, we developed an analysis framework that allows information about OS internals, debug information for the software that is running, etc. to be utilized in an integrated fashion to allow easy interpretation of tainting information. This allows us to trace tainted data to an exact program variable in an application (or the kernel) in the guest, and code propagating tainting to an exact source file and line number.

Our basic usage model consists of two phases. First, we run a simulation in which sensitive data (e.g. coming from the keyboard, network, etc.) is identified as tainted. The workload consists of normal user interaction, e.g. logging into a website via a browser. In the second phase, the simulation data is analyzed with the analysis framework. This allows us to answer open-ended queries about the simulation, such as where tainted data came from, where it was stored, how it was propagated, etc.

We will begin by looking at the implementation of TaintBochs, focusing on modifications to the simulator to facilitate tainting, logging, etc. We will then move on to examine the analysis framework and how it can be used with other tools to gain a complete picture of data lifetime in a system.

3.1 Hardware Level Tainting

There are two central issues to implementing hardware level tainting: first, tracking the location of sensitive state in the system, and, second, deciding how to evolve that state over time to keep a consistent picture of which state is sensitive. We will examine each of these issues in turn.

Shadow Memory To track the location of sensitive data in TaintBochs, we added another memory, set of registers, etc. called a *shadow memory*. The shadow memory tracks taint status of every byte in the system. Every operation performed on machine state by the processor or devices causes a parallel operation to be performed in shadow memory, e.g. copying a word from register A to location B causes the state in the shadow register A to be copied to shadow location B. Thus to determine if a byte is tainted we need only look in the corresponding location in shadow memory.

cted tainted. Maintaining taint status at a byte granularity is a conservative approximation, i.e. we do not ever lose track of sensitive data, although some data may be unnecessarily tainted. Bit granularity would take minimal additional effort, but we have not yet encountered a situation where this would noticeably aid our analysis.

For simplicity, TaintBochs only maintains shadow memory for the guest's main memory and the IA-32's eight general-purpose registers. Debug registers, control registers, SIMD (e.g. MMX, SSE) registers, and flags are disregarded, as is chip set and I/O device state. Adding the necessary tracking for other processor or I/O device state (e.g. disk, frame buffer) would be quite straightforward, but the current implementation is sufficient for many kinds of useful analysis. We are not terribly concerned about the guest's ability to launder taint bits through the processor's debug registers, for example, as our assumption is that software under analysis is not intentionally malicious.

Propagation Policy We must decide how operations in the system should affect shadow state. If two registers A and B are added, and one of them is tainted, is the register where the result are stored also tainted? We refer to the collective set of policies that decide this as the *propagation policy*.

In the trivial case where data is simply copied, we perform the same operation in the address space of shadow memory. So, if the assignment $A \leftarrow B$ executes on normal memory, then $A \leftarrow B$ is also executed on shadow memory. Consequently, if B was tainted then A is now also tainted, and if B was not tainted, A is now also no longer tainted.

The answer is less straightforward when an instruction produces a new value based on a set of inputs. In such cases, our simulator must decide on whether and how to taint the instruction's output(s). Our choices must balance the desire to preserve any possibly interesting taints against the need to minimize spurious reports, i.e. avoid tainting too much data or uninteresting data. This roughly corresponds to the false negatives vs. false positives trade-offs made in other taint analysis tools. As we will see, it is in general impossible to achieve the latter goal perfectly, so some compromises must be made.

Processor instructions typically produce outputs that are some function of their inputs. Our basic propagation policy is simply that *if any byte of any input value is tainted, then all bytes of the output are tainted.* This policy is clearly *conservative* and errs on the side of tainting too much. Interestingly though, with the exception of cases noted below, we have not yet encountered any obviously spurious output resulting from our policy.

> " Our basic propagation policy is simply that if any byte of any input value is tainted, then all bytes of the output are tainted. This policy is clearly conservative and errs on the side of tainting too much."

Chow et al. **Understanding Data Lifetime via Whole System Simulation** USENIX Sec '04

Propagation Problems There are a number of quite common situations where the basic propagation policy presented before either fails to taint interesting information, or taints more than strictly necessary. We have discovered the following so far:

- *Lookup Tables*. Sometimes tainted values are used by instructions as indexes into non-tainted memory (i.e. as an index into a lookup table). Since the tainted value *itself* is not used in the final computation, only the lookup value it points to, the propagation policy presented earlier would not classify the output as tainted.

  This situation arises routinely. For example, Linux routinely remaps keyboard device data through a lookup table before sending keystrokes to user programs. Thus, user programs never directly see the data read in from the keyboard device, only the non-tainted values they index in the kernel's key remapping table.

  Clearly this is not what we want, so we augmented our propagation policy to handle tainted indexes (i.e. tainted pointers) with the following rule: *if any byte of any input value that is involved in the address computation of a source memory operand is tainted, then the output is tainted, regardless of the taint status of the memory operand that is referenced.*
- *Constant Functions*. Tainted values are sometimes used in computations that always produce the same result. We call such computations *constant functions*. An example of such a computation might be the familiar IA-32 idiom for clearing out a register: `xor eax, eax`. After execution of this instruction, `eax` always holds value `0`, regardless of its original value.

  For our purposes, the output of constant functions never pose a security risk, even with tainted inputs, since the input values are not derivable from the output. In the `xor` example above, it is no less the situation as if the programmer had instead written `mov eax, 0`. In the `xor` case, our naive propagation policy taints the output, and in the `mov` case, our propagation policy does not taint the output (since immediate inputs are never considered tainted).

  Clearly, our desire is to never taint the output of constant functions. And while this can clearly be done for special cases like `xor eax, eax` or similar sequences like `sub eax, eax`, this cannot be done in general since the general case (of which the `xor` and `sub` examples are merely degenerate members) is an arbitrary sequence of instructions that ultimately compute a constant function. For example, assuming `eax` is initially tainted, the sequence:

```
mov ebx, eax    ; ebx = eax
shl eax, 1      ; eax = 2 * eax
xor ebx, eax    ; ebx = 0
```

  Always computes (albeit circuitously) zero for `ebx`, regardless of the original value of `eax`. By the time the instruction simulation reaches the `xor`, it has no knowledge of whether its operands have the same value because of some deterministic computation or through simple chance; it must decide, therefore, to taint its output.

  One might imagine a variety of schemes to address this problem. Our approach takes advantage of the semantics of tainted values. For our research, we are interested in tainted data representing secrets like a user-typed password. Therefore, we simply define by fiat that we are only interested in taints on non-zero values. As a result, any operation that produces a zero output value never taints its output, since zero outputs are, by definition, uninteresting.

  This simple heuristic has the consequence that constant functions producing nonzero values can still be tainted. This has not been a problem in practice since constant functions themselves are fairly rare, except for the degenerate ones that clear out a register. Moreover, tainted inputs find their way into a constant function even more rarely, because tainted memory generally represents a fairly small fraction of the guest's overall memory.
- *One-way Functions*. Constant functions are an interesting special case of a more general class of computations we call *one-way functions*. A one-way function is characterized by the fact that its input is not easily derived from its output. The problem with one-way functions is that tainted input values generally produce tainted outputs, just as they did for constant functions. But since the output value gives no practical information about the computation's inputs, it is generally uninteresting to flag such data as tainted from the viewpoint of analyzing information leaks, since no practical security risk exists.

  A concrete example of this situation occurs in Linux, where keyboard input is used as a source of entropy for the kernel's random pool. Data collected into the random pool is passed through various mixing functions, which include cryptographic hashes like SHA-1. Although derivatives of the original keyboard input are used by the kernel when it extracts entropy from the pool, no practical information can be gleaned about the original keyboard input from looking at the random number outputs (at least, not *easily*).

  Our system does not currently try to remove tainted outputs resulting from one-way functions,

" Sometimes tainted values are used by instructions as indexes into non-tainted memory (i.e. as an index into a lookup table)."

For example, Linux routinely remaps keyboard device data through a lookup table before sending keystrokes to user programs. Thus, user programs never directly see the data read in from the keyboard device, only the non tainted values they index in the kernel's key remapping table."

methodology

- shadow memory to track information flow
- time-traveling debugging to uncover relationship between tainted memory

experiment: mozilla

4.1.1 Mozilla

In our first experiment we tracked a user-input password in Mozilla during the login phase of the Yahoo Mail website.

Mozilla was a particularly interesting subject not only because of its real world impact, but also because its size. Mozilla is a massive application (~3.7 million lines of code) written by many different people, it also has a huge number of dependencies on other components (e.g. GUI toolkits).

For our experiment, we began by a booting a Linux[3] guest inside TaintBochs. We then logged in as an unprivileged user, and started X with the `twm` window manager. Inside X, we started Mozilla and brought up the webpage `mail.yahoo.com`, where we entered a user name and password in the login form. Before entering the password, we turned on TaintBoch's keyboard tainting, and afterward we turned it back off. We then closed Mozilla, logged out, and closed TaintBochs.

- *Kernel random number generator.* The Linux kernel has a subsystem that generates cryptographically secure random numbers, by gathering and mixing entropy from a number of sources, including the keyboard. It stores keyboard input temporarily in a circular queue for later batch processing. It also uses a global variable `last_scancode` to keep track of the previous key press; the keyboard driver also has a similar variable `prev_scancode`.
- *XFree86 event queue.* The X server stores user-input events, including keystrokes, in a circular queue for later dispatch to X clients.
- *Kernel socket buffers.* In our experiment, X relays keystrokes to Mozilla and its other clients over Unix domain sockets using the `writev` system call. Each call causes the kernel to allocate a `sk_buff` socket structure to hold the data.
- *Mozilla strings.* Mozilla, written in C++, uses a number of related string classes to process user data. It makes no attempt to curb the lifetime of sensitive data.
- *Kernel tty buffers.* When the user types keyboard characters, they go into a `struct tty_struct` "flip buffer" directly from interrupt context. (A flip buffer is divided into halves, one used only for reading and the other used only for writing. When data that has been written must be read, the halves are "flipped" around.) The key codes are then copied into a tty, which X reads.

Slowinska & Bos: **Pointless Tainting? Evaluating the Practicality of Pointer Tainting** Eurosys '09

Pointless Tainting? Evaluating the Practicality of Pointer Tainting

Asia Slowinska
Vrije Universiteit Amsterdam
asia@few.vu.nl

Herbert Bos
Vrije Universiteit Amsterdam and NICTA *
herbertb@cs.vu.nl

Abstract

This paper evaluates pointer tainting, an incarnation of Dynamic Information Flow Tracking (DIFT), which has recently become an important technique in system security. Pointer tainting has been used for two main purposes: detection of privacy-breaching malware (e.g., trojan keyloggers obtaining the characters typed by a user), and detection of memory corruption attacks against non-control data (e.g., a buffer overflow that modifies a user's privilege level). In both of these cases the attacker does not modify control data such as stored branch targets, so the control flow of the target program does not change. Phrased differently, in terms of instructions executed, the program behaves 'normally'. As a result, these attacks are exceedingly difficult to detect. Pointer tainting is considered one of the only methods for detecting them in unmodified binaries. Unfortunately, almost all of the incarnations of pointer tainting are flawed. In particular, we demonstrate that the application of pointer tainting to the detection of keyloggers and other privacy-breaching malware is problematic. We also discuss whether pointer tainting is able to reliably detect memory corruption attacks against non-control data. We found that pointer tainting generates itself the conditions for false positives. We analyse the problems in detail and investigate various ways to improve the technique. Most have serious drawbacks in that they are either impractical (and incur many false positives still), and/or cripple the technique's ability to detect attacks. In conclusion, we argue that depending on architecture and operating system, pointer tainting may have some value in detecting memory corruption attacks (albeit with false negatives and not on the popular x86 architecture), but it is fundamentally not suitable for automated detecting of privacy-breaching malware such as keyloggers.

Categories and Subject Descriptors D.4.6 [*Security and Protection*]: Invasive software

General Terms Security, Experimentation

Keywords dynamic taint analysis, pointer tainting

1. Introduction

Exploits and trojans allow attackers to compromise machines in various ways. One way to exploit a machine is to use techniques like buffer overflows or format string attacks to divert the flow of execution to code injected by the attacker. Alternatively, the same exploit techniques may attack non-control data [Chen 2005b]; for instance a buffer overflow that modifies a value in memory that represents a user's identity, a user's privilege level, or a server configuration string. Non-control data attacks are even more difficult to detect than attacks that divert the control flow. After all, the program does not execute any foreign code, does not jump to unusual places, and does not exhibit strange system call patterns or any other tell-tale signs that indicate that something might be wrong.

While protection for some of these attacks may be provided if we write software in type-safe languages [Jim 2002], compile with specific compiler extensions [Castro 2006, Akritidis 2008], or verify with formal methods [Elphinstone 2007], much of the system software in current use is written in C or C++ and often the source of the software is not available, and recompilation is not possible.

Worse, even with the most sophisticated languages, it is difficult to stop users from installing trojans. Often trojans masquerade as useful programs, like pirated copies of popular applications, games, or 'security'-tools, with keylogging, privacy theft and other malicious activities as hidden features. No exploit is needed to compromise the system at all. Once inside, the malware may be used to join a spam botnet, damage the system, attack other sites, or stealthily spy on a user. Again, stealthy spies are harder to detect than 'loud'

* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

EuroSys '09, 1–3, April 2009, Nuremberg, Germany.

Slowinska & Bos: **Pointless Tainting? Evaluating the Practicality of Pointer Tainting**

Eurosys '09

programs that damage systems, or engage in significant network activity. The trojan spyware, installed by the user, may use legitimate APIs to obtain and store the characters that are typed in by the users (or data in files, buffers, or on the network). From a system's perspective, the malware is not doing anything 'wrong'.

In light of the above, we distinguish between attacks that divert the control flow of a program and those that do not. *Control diversion* typically means that a pointer in a process is manipulated by an attacker so that when it is dereferenced, the program starts executing instructions different from the ones it would normally execute at that point. *Non control diverting* attacks, on the other hand, include memory corruption attacks against non-control data and privacy breaching malware like keyloggers and sniffers. Memory corruption attacks against non-control data manipulate data values that are not directly related to the flow of control; for instance, a value that represents a user's privilege level, or the length in bytes of a reply buffer. The attack itself does not lead to unusual code execution. Rather, it leads to elevated privileges, or unusual replies. The same is true for privacy breaching malware like sniffers and trojan keyloggers.

Pointer tainting as advertised is attractive. It is precisely these difficult to detect, stealthy non-control-diverting attacks that are the focus of pointer tainting [Chen 2005a]. At the same time, the technique works against control-diverting attacks also. We will discuss pointer tainting in more detail in later sections. For now, it suffices to define it as a form of dynamic information flow tracking (DIFT) [Suh 2004] which marks the origin of data by way of a taint bit in a shadow memory that is inaccessible to software. By tracking the propagation of tainted data through the system (e.g., when tainted data is copied, but also when tainted pointers are dereferenced), we see whether any value derived from data from a tainted origin ends up in places where it should never be stored. For instance, we shall see that some projects use it to track the propagation of keystroke data to ensure that untrusted and unauthorised programs do not receive it [Yin 2007]. By implementing pointer tainting in hardware [Dalton 2007], the overhead is minimal.

Pointer tainting is very popular because (a) it can be applied to unmodified software without recompilation, and (b) according to its advocates, it incurs hardly (if any) false positives, and (c) it is assumed to be one of the only (if not *the* only) reliable techniques capable of detecting both control-diverting and non-control-diverting attacks without requiring recompilation. Pointer tainting has become a unique and extremely valuable detection method especially due to its presumed ability to detect non-control-diverting attacks. As mentioned earlier, non-control-diverting attacks are more worrying than attacks that divert the control flow, because they are harder to detect. Common protection mechanisms like address space randomisation and stackguard [Bhatkar 2005, Cowan 1998] present in several modern operating systems are ineffective against this type of attack. The same is true for almost all forms of system call monitoring [Provos 2003, Giffin 2004]. As a result, some trojan keyloggers have been active for years (often undetected). In one particularly worrying case, a keylogger harvested over 500,000 login details for online banking and other accounts [Raywood 2008]. At the same time, the consequences of a successful non-control-diverting attack may be as severe as with a control-diverting attack. For instance, passwords obtained by a keylogger often give attackers full control of the machines. The same is true for buffer overflows that modify a user's privilege level.

However, pointer tainting is not working as advertised. Inspired by a string of publications about pointer tainting in top venues [Chen 2005a;b, Yin 2007, Egele 2007, Dalton 2007, Yin 2008, Venkataramani 2008, Dalton 2008], several of which claim zero false positives, we tried to build a keylogger detector by means of pointer tainting. However, what we found is that for privacy-breaching malware detection, the method is flawed. It incurs both false positives and negatives. The false positives appear particularly hard to avoid. There is no easy fix. Further, we found that almost all existing applications of pointer tainting to detection of memory corruption attacks are also problematic, and none of them are suitable for the popular x86 architecture and Windows operating system.

In this paper, we analyse the fundamental limitations of the method when applied to detection of privacy-breaching malware, as well as the practical limitations in current applications to memory corruption detection. Often, we will see that the reason is that 'fixing the method is breaking it': simple solutions to overcome the symptoms render the technique vulnerable to false positives or false negatives.

Others have discussed minor issues with projects that use pointer tainting [Dalton 2006], and most of these have been addressed in later work [Dalton 2008]. To the best of our knowledge, nobody has investigated the technique in detail, nobody has shown that it does not work against keyloggers, and we are the first to report the complicated problems with the technique that are hard to overcome. We are also the first to evaluate the implications experimentally.

In summary, the contributions of this paper are:

1. an in-depth analysis of the problems of pointer tainting on real systems which shows that it does not work against malware spying on users' behaviour, and is problematic in other forms also;
2. an analysis and evaluation of all known fixes to the problems that shows that they all have serious shortcomings.

We emphasise that this paper is not meant as an attack on existing publications. In our opinion, previous papers underestimated the method's problems. We hope that our work will help others avoid making the mistakes we made

user. Again, stealthy spies are harder to detect than 'loud'

"…in-depth analysis of the problems of pointer tainting on real systems which shows that it does not work against malware spying on users' behaviour"

"an analysis and evaluation of all known fixes to the problems that shows that they all have serious shortcomings."

Slowinska & Bos: **Pointless Tainting? Evaluating the Practicality of Pointer Tainting** Eurosys '09

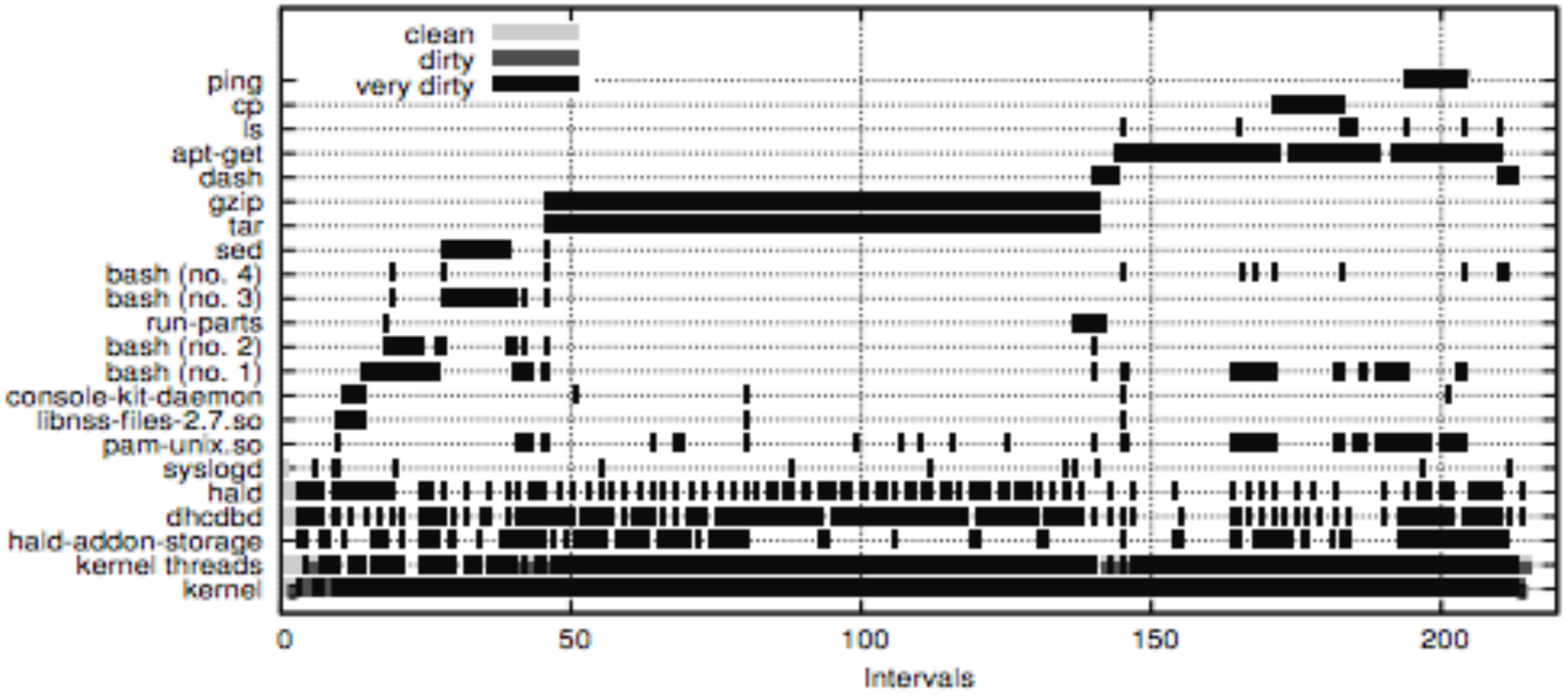


Figure 2. The taintedness of the processes constituting 90% of all context switches. In this and all similar plots the following explanation holds. The x-axis is divided into scheduling intervals, spanning 50 scheduling operations each. Time starts when taint is introduced in the system. In an interval, several processes are scheduled. For each of these, we take a random sample from the interval to form a datapoint. So, even if `gzip` is scheduled multiple times in an interval, it has only one datapoint. A datapoint consists of two small boxes drawn on top of each other, separated by a thin white line. The smaller one at the top represents the taintedness of ebp and esp. The bottom, slightly larger one represents all other registers. We use three colours: lightgrey means the registers are clean, darkgrey means less than half of considered registers are tainted, and black means that half or more are tainted (very dirty). Absence of a box means the process was not scheduled.

Analyzing Concurrent Systems

King et al. **Debugging Operating Systems with Time-Traveling Virtual Machines**

Usenix '05

3 Time-traveling virtual machines

A time-traveling virtual machine should have two capabilities. First, it should be able to reconstruct the complete state of the virtual machine at any point in a run, where a run is defined as the time from when the virtual machine was powered on to the last instruction it executed. Second, it should be able to start from any point in a run and from that point replay the same instruction stream that was executed during the original run from that point. This section describes how TTVM achieves these capabilities through a combination of logging, replay, and checkpointing.

3.1 Logging and replaying a VM

The foundational capability in TTVM is the ability to replay a run from a given point in a way that matches the original run instruction for instruction. Replay causes the virtual machine to transition through the same states as it went through during the original run; hence replay enables one to reconstruct the complete state of the virtual machine at any point in the run. TTVM uses the ReVirt logging/replay system to provide this capability [9]. This section briefly summarizes how ReVirt logs and replays the execution of a virtual machine.

A virtual machine can be replayed by starting from a checkpoint, then replaying all sources of nondeterminism [5, 9]. For UML, the sources of nondeterminism are external input from the network, keyboard, and real-time clock and the timing of virtual interrupts. The VMM replays network and keyboard input by logging the calls that read these devices during the original run and regenerating the same data during the replay run. Likewise, we configure the CPU to cause reads of the real-time clock to trap to the VMM, where they can be logged or regenerated.

To replay a virtual interrupt, ReVirt logs the instruction in the run at which it was delivered and re-delivers the interrupt at this instruction during replay. This point is identified uniquely by the number of branches since the start of the run and the address of the interrupted instruction [19]. ReVirt uses a performance counter on the Intel Pentium 4 CPU to count the number of branches during logging, and it uses the same performance counter and instruction breakpoints to stop at the interrupted instruction during replay. Replaying interrupts enables ReVirt to replay the scheduling order of multi-threaded guest operating systems and applications, as long as the VMM exports the abstraction of a uniprocessor virtual machine [22]. Researchers are investigating ways to support replay on multiprocessors [29].

3.2 Host device drivers in the guest OS

In general, VMMs export a limited set of virtual devices. Some VMMs export virtual devices that exist in hardware (e.g., VMware Workstation exports an emulated AMD Lance Ethernet card); others (like UML) export virtual devices that have no hardware equivalent. Exporting a limited set of virtual devices to the guest OS is usually considered a benefit of virtual-machine systems, because it frees guest OSs from needing device drivers for myriad host devices [26]. However, when using virtual machines to debug operating systems, the limited set of virtual devices prevents programmers from using and debugging drivers for real devices; programmers can only debug the architecture-independent portion of the guest OS. There are two ways to address this limitation and enable the programmer to run and debug real device drivers in a guest OS. With both strategies, real device drivers can be included in the guest OS without being modified or re-compiled.

The first way to run a real device driver in the guest OS is for the VMM to provide a software emulator for that device. The device driver issues the normal set of I/O instructions: `IN`/`OUT` instructions, memory-mapped I/O, DMA commands, and interrupts. The VMM traps these privileged instructions and forwards them to/from the software device emulator. With this strategy, ReVirt can log and replay device driver code in the same way it logs and replays the rest of the guest OS. If one runs the VMM's software device emulator above ReVirt's logging system (and above the checkpoint system described in Section 3.3), ReVirt will guide the emulator and device driver code through the same instruction sequence during replay as they executed during logging. While this first strategy fits in well with the existing ReVirt system, it only works if one has an accurate software emulator for the device whose driver one wishes to debug.

We modified UML to provide a second way to run real device drivers in the guest OS, which works even when no software emulator exists for the device of interest. With this strategy, the VMM traps and forwards the privileged I/O instructions and DMA requests issued by the guest OS device driver to the actual hardware. The programmer specifies which devices UML can access, and the VMM enforces the proper I/O port space and memory access for the device.

This second strategy requires extensions to enable ReVirt to log and replay the execution of the device driver. Whereas the first strategy placed the device emulator above the ReVirt logging layer, the second strategy forwards driver actions to the actual hardware device. Because this device may not be deterministic, ReVirt must log any information sent from the device to the driver. Specifically, ReVirt must log and replay the data returned

"Replay causes the virtual machine to transition through the same states as it went through during the original run"

"A virtual mac… replayed by sta… checkpoint, then replaying all sources of nondeterminism [...] the network, keyboard, clock, and timing of interrupts"

Sequential

problem statement

“how to build a dynamic analysis tool that correctly runs in a concurrent environment?”

“What challenges does concurrent software have that dynamic analysis could help with?”

concurrent record & replay

- use locking operations as sequencing points
- use virtual memory page protection
- (come up with ideas for custom hardware)

Nethercote et al. **Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation**

PLDI '07

tions. Valgrind provides an *events system* to describe such changes.

Let us first consider the accesses done by system calls. All system calls access registers: they read their arguments from registers and/or memory, and they write their return value to a register. Many system calls also access user-mode memory via pointer arguments, e.g. `settimeofday` is passed pointers to two structs which it reads from, and `gettimeofday` fills in two structs with data. Knowing which registers and memory locations are accessed by every system call is difficult because there are many system calls (around 300 for Linux), some of which have tens or hundreds of sub-cases, and there are many differences across platforms. Several things must be known for each system call: how many arguments it takes, each argument's size, which ones are pointers (and which of those can be `NULL`), which ones indicate buffer lengths, which ones are null-terminated strings, which ones are not read in certain cases (e.g. the third argument of `open` is only read if the second argument has certain values), and the sizes of various types (e.g. `struct timeval` used by `gettimeofday` and `settimeofday`).

Valgrind does not encode this information about system calls in its IR, because there are too many system calls and too much variation across platforms to do so cleanly. Instead it provides the events system to inform tools about register and memory accesses in tricky cases, with a small amount of help from the programmer all stack switches can be detected.

The remaining events in Table 1 inform tools about allocations done at program start-up and via system calls.

3.13 Function Replacement and Function Wrapping

Valgrind supports *function replacement*, i.e. it allows a tool to replace any function in a program with an alternative function. A replacement function can also call the function it has replaced. This allows *function wrapping*, which is particularly useful for inspecting the arguments and return value of a function.

3.14 Threads

Threads pose a particular challenge for shadow value tools. The reason is that loads and stores become non-atomic: each load/store translates into the original load/store plus a shadow load/store. On a uni-processor machine, a thread switch might occur between these two operations. On a multi-processor machine, concurrent memory accesses to the same memory location may complete in a different order to their corresponding shadow memory accesses. It is unclear how to best deal with this, as a fine-grained locking approach would likely be slow.

To sidestep this problem, Valgrind serialises thread execution with a thread locking mechanism. Only the thread holding the lock can run, and threads drop the lock before they call a blocking system call,[6] or after they have been running for a while (100,000 code blocks). The lock is implemented using a pipe which holds a single character; each thread tries to read the pipe, only one thread will be successful, and the others will block until the running thread relinquishes the lock by putting a character back in the pipe. Thus the kernel still chooses which thread is to run next, but Valgrind dictates when thread-switches occur and prevents more than one thread from running at a time.

This is the third thread serialisation mechanism that has been used in Valgrind, and is by far the most robust. The first one [18, 15] involved Valgrind providing a serialised version of the `libpthread` library. This only worked with programs using pthreads. It also caused many problems because on Linux systems, glibc and the pthreads library are tightly bound and interact in various ways "under the covers" that are difficult to replicate.[7] The second one was more like the current one, but was more complex, requiring extra kernel threads to cope with blocking I/O.

This serialisation is a unique Valgrind feature not shared by other DBI frameworks. It has both pros and cons: it means that Valgrind tools using shadow memory can ignore the atomicity issue. However, as multi-processor machines become more popular, the resulting performance shortcomings for multi-threaded programs will worsen. How to best overcome this problem remains an open research question.

3.15 Signals

[5] In comparison, chaining improved Strata's basic slow-down factor from 22.1x to 4.1x, because dispatching takes about 250 cycles [24]. Valgrind's slow-down even without chaining is 4.3x.

Req.	Valgrind events	Called from	Memcheck callbacks
R4	`pre_reg_read`, `post_reg_write`	Every system call wrapper	`check_reg_is_defined`, `make_reg_defined`
	`pre_mem_read{,_asciiz}`	Many system call wrappers	`check_mem_is_defined{,_asciiz}`
	`pre_mem_write`, `post_mem_write`	Many system call wrappers	`check_mem_is_addressable`, `make_mem_defined`
R5	`new_mem_startup`	Valgrind's code loader	`make_mem_defined`
R6	`new_mem_mmap`, `die_mem_munmap`	`mmap` wrapper, `munmap` wrapper	`make_mem_defined`, `make_mem_noaccess`
	`new_mem_brk`, `die_mem_brk`	`brk` wrapper	`make_mem_undefined`, `make_mem_noaccess`
	`copy_mem_mremap`	`mremap` wrapper	`copy_range`
R7	`new_mem_stack`, `die_mem_stack`	Instrumentation of SP changes	`make_mem_undefined`, `make_mem_noaccess`

Table 1. Valgrind events, their trigger locations, and Memcheck's callbacks for handling them.

the hash is recomputed and checked, and if it does not match, the block is discarded and the code retranslated.

This has a high run-time cost. Therefore, by default Valgrind only uses this mechanism for code that is on the stack. This is enough to handle the trampolines that some compilers (e.g. GCC) put on the stack when running nested functions, which we have found to be the main cause of self-modifying code.[8] This minimises the cost, as only code on the stack is slowed down. The mechanism can also be turned off altogether or turned on for every block.

Valgrind also provides another mechanism for handling self-modifying code—a client request which tells it to discard any translations of instructions in a certain address range. It is most useful for dynamic code generators such as JIT compilers.

4. Valgrind's Shadow Value Support

This section describes how the features described in the previous section support all nine shadow value requirements. Because these requirements are a superset of most DBA tools' requirements, Valgrind supports most conceivable DBA tools.

R1: Provide shadow registers. Valgrind has three noteworthy features that make shadow registers easy to use. First, ***shadow registers are first-class entities:*** (a) space is provided for them in the ThreadState, (b) they can be accessed just as easily as guest registers, (c) they can be manipulated and operated on in the same ways. This makes complex shadow operations code natural and

"Valgrind serialises thread execution with a thread locking mechanism. Only the thread holding the lock can run,"

"The kernel still chooses which thread is to run next, but Valgrind dictates when thread-switches occur and prevents more than one thread from running at a time."

"How to best overcome this problem remains an open research question."

Savage et al: **Eraser: A Dynamic Data Race Detector for Multithreaded Programs**

SOSP '97

Eraser: A Dynamic Data Race Detector for Multithreaded Programs

STEFAN SAVAGE
University of Washington
MICHAEL BURROWS, GREG NELSON, and PATRICK SOBALVARRO
Digital Equipment Corporation
and
THOMAS ANDERSON
University of California at Berkeley

Multithreaded programming is difficult and error prone. It is easy to make a mistake in synchronization that produces a data race, yet it can be extremely hard to locate this mistake during debugging. This article describes a new tool, called Eraser, for dynamically detecting data races in lock-based multithreaded programs. Eraser uses binary rewriting techniques to monitor every shared-memory reference and verify that consistent locking behavior is observed. We present several case studies, including undergraduate coursework and a multithreaded Web search engine, that demonstrate the effectiveness of this approach.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*; D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids*; *monitors*; *tracing*; D.4.1 [**Operating Systems**]: Process Management—*concurrency*; *deadlock*; *multiprocessing/multiprogramming*; *mutual exclusion*; *synchronization*

General Terms: Algorithms, Experimentation, Reliability

Additional Key Words and Phrases: Binary code modification, multithreaded programming, race detection

1. INTRODUCTION

Multithreading has become a common programming technique. Most commercial operating systems support threads, and popular applications like Microsoft Word and Netscape Navigator are multithreaded.

An earlier version of this article appeared in the *Proceedings of the 16th ACM Symposium on Operating System Principles*, St. Malo, France, 1997.
Authors' addresses: S. Savage and T. Anderson, Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98195; email: savage@cs.washington.edu; M. Burrows, G. Nelson, and P. Sobalvarro, Systems Research Center, Digital Equipment Corporation, 130 Lytton Avenue, Palo Alto, CA 94301.

Savage et al: **Eraser: A Dynamic Data Race Detector for Multithreaded Programs**

SOSP '97

On each read of v by thread t,
  set $C(v) := C(v) \cap locks\_held(t)$;
  if $C(v) := \{\ \}$, then issue a warning.
On each write of v by thread t,
  set $C(v) := C(v) \cap write\_locks\_held(t)$;
  if $C(v) = \{\ \}$, then issue a warning.

That is, locks held purely in read mode are removed from the candidate set when a write occurs, as such locks held by a writer do not protect against a data race between the writer and some other reader thread.

3. IMPLEMENTING ERASER

Eraser is implemented for the Digital Unix operating system on the Alpha processor, using the ATOM [Srivastava and Eustace 1994] binary modification system. Eraser takes an unmodified program binary as input and adds instrumentation to produce a new binary that is functionally identical, but includes calls to the Eraser runtime to implement the Lockset algorithm.

To maintain $C(v)$, Eraser instruments each load and store in the program. To maintain $lock\_held(t)$ for each thread t, Eraser instruments each call to acquire or release a lock, as well as the stubs that manage thread initialization and finalization. To initialize $C(v)$ for dynamically allocated data, Eraser instruments each call to the storage allocator.

Eraser treats each 32-bit word in the heap or global data as a possible shared variable, since on our platform a 32-bit word is the smallest memory-coherent unit. Eraser does not instrument loads and stores whose address mode is indirect off the stack pointer, since these are assumed to be stack references, and shared variables are assumed to be in global locations or in the heap. Eraser will maintain candidate sets for stack locations that are accessed via registers other than the stack pointer, but this is an artifact of the implementation rather than a deliberate plan to support programs that share stack locations between threads.

When a race is reported, Eraser indicates the file and line number at which it was discovered and a backtrace listing of all active stack frames. The report also includes the thread ID, memory address, type of memory access, and important register values such as the program counter and stack pointer. When used in conjunction with the program's source code, we have found that this information is usually sufficient to locate the origin of the race. If the cause of a race is still unclear, the user can direct Eraser to log all the accesses to a particular variable that result in a change to its candidate lock set.

"Eraser instruments each call to acquire or release a lock"

Dunlap et al: **Execution Replay for Multiprocessor Virtual Machines**

VEE '08

Execution Replay for Multiprocessor Virtual Machines

George W. Dunlap, Dominic G. Lucchetti, Peter M. Chen
Electrical Engineering and Computer Science Dept.
University of Michigan
Ann Arbor, MI 48109-2122
{dunlapg,dlucchet,pmchen}@umich.edu

Michael A. Fetterman
University of Cambridge Computer Laboratory
15 JJ Thompson Avenue, Cambridge, UK, CB3 0FD
Michael.Fetterman@cl.cam.ac.uk

Abstract

Execution replay of virtual machines is a technique which has many important applications, including debugging, fault-tolerance, and security. Execution replay for single processor virtual machines is well-understood, and available commercially. With the advancement of multi-core architectures, however, multiprocessor virtual machines are becoming more important. Our system, SMP-ReVirt, is the first system to log and replay a multiprocessor virtual machine on commodity hardware. We use hardware page protection to detect and accurately replay sharing between virtual cpus of a multi-cpu virtual machine, allowing us to replay the entire operating system and all applications. We have tested our system on a variety of workloads, and find that although sharing under SMP-ReVirt is expensive, for many workloads and applications, including debugging, the overhead is acceptable.

Categories and Subject Descriptors C.4 [*Computer Systems Organization*]: Performance of Systems — Measurement Techniques; D.4.1 [*Operating Systems*]: Process Management — multiprocessing

General Terms Design, Measurement, Performance, Reliability, Security

Keywords ReVirt, execution replay, multithreading, determinism, race recording, multiprocessors, virtual machines, Xen, direct memory access, SPLASH, page protections

1. Introduction

Execution replay gives the ability to reconstruct the past execution of a system. In conjunction with a checkpoint of the system state, it gives the ability to reconstruct the entire state at any point in time over the replay interval. This ability is useful for several different applications. For debugging, it allows a programmer to inspect the execution and state of a particular run of a system, even in the face of non-determinism[9, 14, 5, 8]. For security, it allows a system administrator to go back and inspect the entire state of the system before, during, and after an attack, allowing the system administrator to determine how the attack took place and observe the attacker's activities[6]. For fault tolerance, execution replay allows the state of a system just before a crash to be recovered without the need for frequent checkpoints[7, 4, 11]. Recent work has also used execution replay to efficiently collect and store software architectural traces[19].

A simple way to apply execution replay to a wide range of software is to implement execution replay for virtual machines[4]. Running software in a virtual machine capable of being replayed allows a user to take advantage of execution replay without needing to modify software running inside the virtual machine. It also has the advantage of being able to use execution replay on an operating system kernel.

In order to implement execution replay in a virtual machine, any non-deterministic event that affects the virtual machine's state must be recorded. This state includes all memory allocated to the virtual machine, the processor registers, and the disk. For a single processor system, non-deterministic events include any external input (such as keyboard, mouse, or network), as well as the timing of non-deterministic events like interrupts. The techniques for replaying single processor systems are well understood, and are even available commercially[19].

With the increasing prevalence of multi-core processors, execution replay on multiprocessor systems has become more important. Implementing replay for multiprocessor systems is much more challenging than single processor systems, however. Because writes on one processor can affect reads on another processor, the results of memory races must be recorded and replayed. Existing solutions require modification to software, or massive modifications to hardware.

We have built a system, SMP-ReVirt, which is the first system to log and replay multiprocessor virtual machines on commodity hardware. In order to detect and replay the results of memory races, we use hardware page protections, available on all modern desktop and server processors. This technique allows us to log and replay unmodified multiprocessor systems, including multiprocessor kernels running inside of a virtual machine.

Logging makes sharing more expensive, but the end-to-end impact on performance varies widely depending on the workload. For some applications it is prohibitively expensive, while for others there is little impact.

This paper explores execution replay for multiprocessor virtual machines. Section 2 introduces the basic concepts, terms, and requirements of execution replay for single processor virtual machines. It then discusses the complications that shared-memory systems introduce, and describes techniques to address them. Section 3 describes the research prototype we built using the Xen hypervisor, describing the implementation of the general principles in more detail, and describing some of the technical issues involved. In Section 4, we evaluate our research prototype, investigating the

VEE'08, March 5–7, 2008, Seattle, Washington, USA.

Dunlap et al: **Execution Replay for Multiprocessor Virtual Machines**

VEE '08

If b and c are unrelated, we say that the constraints above are *over-constrained*, because they cause the replay system to run more strictly than necessary: either P_2 must wait until P_1 reaches b (although the data was ready at a), or P_2 stops and waits at c (although it is not necessary to stop and wait until d), or both. Over-constraining reduces the potential parallelism during a replay run, but can be taken advantage of to reduce the number of constraints or simplify logging.

Suppose instead that a and d are writes to one area of memory, and b and c were writes to a second area of memory. In this case, $b \rightarrow c$ would be a necessary constraint. However, the constraint $a \rightarrow d$ would be *redundant*, because the ordering $a \rightarrow d$ it is implied by the constraint $b \rightarrow c$. Removing redundant constraints can decrease the log size.

A logging system for a shared memory system must generate a set of constraints that will satisfy the ordering requirement, but is free to choose any set of constraints that will meet that ordering.

To detect which memory operations need to be ordered, we implement a *concurrent-read, exclusive-write* (CREW) protocol between *virtual cpus* in a multiprocessor virtual machine. This technique for detecting constraints was first introduced by [9]. The CREW protocol stipulates that each shared object may be in one of the following two states:

- *concurrent-read*: All cpus have read permission, but none have write permission.
- *exclusive-write*: One cpu (called the *owner*) has both read and write permission; all virtual cpus have no permission.

Each read or write operation to shared memory is checked for access before executing. If a virtual cpu attempts a memory operation for which it has insufficient access, the CREW system must make requests to the other processors to decrease their permissions, so that it may increase its own. We call these increases and decreases in permissions *CREW events*.

The CREW protocol has the following property: if two memory instructions on different processors access the same page, and one of them is a write, there will be a CREW event between the instructions on each processor. This corresponds precisely with the ordering requirement. We can take advantage of this property to detect potential races and generate constraints sufficient to replay the order of accesses for a given execution.

In order to check for access of shared memory reads and writes, we use hardware page protections, available on all modern desktop processors. Hardware page protections are enforced by the *memory management unit*(MMU), which will check each read and write as the instruction executes, and cause a fault to the hypervisor on any violation. Because the checks are done in hardware, the common case is very fast. It also allows us to interpose on reads and writes without modifying the software running on the guest.

Generating constraints from CREW events is straightforward. Each CREW fault will cause two CREW events: a privilege increase on one processor, and a privilege decrease on another processor. If a is the point of privilege decrease, and b is the point of privilege increase, then the constraint $a \rightarrow b$ will be sufficient to order any reads and writes associated with this CREW event.

To see why this is so, consider a particular interaction between two processors, P_1 and P_2. Suppose that instruction b at P_2 writes to a page which is in concurrent-read mode. This instruction will cause a fault into the CREW subsystem. The CREW system will then reduce the privileges of P_1, and increase the privileges of P_2. Let us call the instruction this privilege-decrease happens instruction a. Processor P_1 has had read permission from the time it received permission through point a. Any instruction during that time may have read the page that b is about to write. We cannot tell when the last access was using page protections, but we know that it will be before a by program order, so constraining $a \rightarrow b$ will give us the ordering we need.

Constraining on privilege-reduction events rather than on the last read does mean that our replay will be over-constrained. This is because we do not have access to when the last read or write to the page actually occurred; any instruction executed before the privilege-reduction event could have accessed the page.

2.2 Direct Memory Access

Modern hardware systems allow physical devices to write directly to main memory, without involving the processor. This is called *direct memory access* (DMA). DMA eliminates the overhead of the processor copying data from the device to memory.

Replaying DMA presents some difficulties. In DMA, a device acts as another processor with respect to memory transactions. A single processor system with DMA-enabled devices is effectively a multiprocessor system from replay's perspective. However, unlike peer processors in an SMP system, the devices do not have an MMU that we can use to interpose on accesses[3]. How are we to involve devices in the CREW protocol?

The key observation is that DMA devices are not generally self-motivated peers. They only write to memory in response to a request from a cpu. Requests typically follow a *transaction* model, where a cpu will specify an operation and an area of memory. The device will access the memory during the operation, and inform the cpu when the operation is completed. After the transaction is finished, the device will not access to the memory again. While this transaction is taking place, it is generally not correct for the cpu to access the memory assigned to the device to do DMA.

If the device follows this type of transaction model, where the device will only access memory between certain well-defined boundaries, and the cpu does not need to access memory to the device until a transaction is completed, and if the hypervisor can interpose and understand the commands from the guest to the device and the device's responses, we can model the device as a *non-preemptible actor* in the CREW protocol. A non-preemptible actor does explicit acquire and release of pages before and after a transaction, rather than acquiring them on demand and having them preempted, as preemptible actors such as virtual cpus do. When a cpu issues a DMA command to the device, the hypervisor informs the CREW protocol, which acquires the appropriate privileges on behalf of the device (either concurrent-read or exclusive-write, depending on the transaction). When the device informs the cpu that the transaction is done, the hypervisor informs the CREW protocol, which will release access on behalf of the device.

If any virtual cpu tries to access a page in a way that is incompatible with the CREW privileges of some device on a system, the CREW system must block its execution until the device has finished the transaction associated with that page. In this way, we do not need to rely on the correctness of kernels or device drivers inside the virtual machine; only on the correctness of the hardware.

During replay, the constraint replay system must ensure that the DMA is replayed at the proper time with respect to the other processors. If we do not use the device during replay, we must log the data from the DMA during logging in order to replay it from the log during replay; otherwise, we must ensure that the device does the DMA properly.

[3] Some new systems include an IO-MMU, for controlling DMA access to memory. However, these systems are designed to prevent buggy drivers and devices from corrupting system state, and do not necessarily provide ways to continue an interrupted operation after a fault. Re-executing a faulting operation is fundamental to our technique.

- **concurrent-read**: All cpus have read permission, but none have write permission.
- **exclusive-write**: One cpu (called the owner) has both read and write permission; all virtual cpus have no permission.

"if two memory instructions on different processors access the same page, and one of them is a write, there will be a CREW event between the instructions on each processor."

Dunlap et al: **Execution Replay for Multiprocessor Virtual Machines**

VEE ‘08

If b and c are unrelated, we say that the constraints above are *over-constrained*, because they cause the replay system to run more strictly than necessary: either P_2 must wait until P_1 reaches b (although the data was ready at a), or P_2 stops and waits at c (although it is not necessary to stop and wait until d), or both. Over-constraining reduces the potential parallelism during a replay run, but can be taken advantage of to reduce the number of constraints or simplify logging.

Suppose instead that a and d are writes to one area of memory, and b and c were writes to a second area of memory. In this case, $b \rightarrow c$ would be a necessary constraint. However, the constraint $a \rightarrow d$ would be *redundant*, because the ordering $a \rightarrow d$ it is implied by the constraint $b \rightarrow c$. Removing redundant constraints can decrease the log size.

A logging system for a shared memory system must generate a set of constraints that will satisfy the ordering requirement, but is free to choose any set of constraints that will meet that ordering.

To detect which memory operations need to be ordered, we implement a *concurrent-read, exclusive-write* (CREW) protocol between *virtual cpus* in a multiprocessor virtual machine. This technique for detecting constraints was first introduced by [9]. The CREW protocol stipulates that each shared object may be in one of the following two states:

- *concurrent-read*: All cpus have read permission, but none have write permission.
- *exclusive-write*: One cpu (called the *owner*) has both read and write permission; all virtual cpus have no permission.

Each read or write operation to shared memory is checked for access before executing. If a virtual cpu attempts a memory operation for which it has insufficient access, the CREW system must make requests to the other processors to decrease their permissions, so that it may increase its own. We call these increases and decreases in permissions *CREW events*.

The CREW protocol has the following property: if two memory instructions on different processors access the same page, and one of them is a write, there will be a CREW event between the instructions on each processor. This corresponds precisely with the ordering requirement. We can take advantage of this property to detect potential races and generate constraints sufficient to replay the order of accesses for a given execution.

In order to check for access of shared memory reads and writes, we use hardware page protections, available on all modern desktop processors. Hardware page protections are enforced by the *memory management unit*(MMU), which will check each read and write as the instruction executes, and cause a fault to the hypervisor on any violation. Because the checks are done in hardware, the common case is very fast. It also allows us to interpose on reads and writes without modifying the software running on the guest.

Generating constraints from CREW events is straightforward. Each CREW fault will cause two CREW events: a privilege increase on one processor, and a privilege decrease on another processor. If a is the point of privilege decrease, and b is the point of privilege increase, then the constraint $a \rightarrow b$ will be sufficient to order any reads and writes associated with this CREW event.

To see why this is so, consider a particular interaction between two processors, P_1 and P_2. Suppose that instruction b at P_2 writes to a page which is in concurrent-read mode. This instruction will cause a fault into the CREW subsystem. The CREW system will then reduce the privileges of P_1, and increase the privileges of P_2. Let us call the instruction this privilege-decrease happens instruction a. Processor P_1 has had read permission from the time it received permission through point a. Any instruction during that time may have read the page that b is about to write. We cannot tell when the last access was using page protections, but we know that it will be before a by program order, so constraining $a \rightarrow b$ will give us the ordering we need.

Constraining on privilege-reduction events rather than on the last read does mean that our replay will be over-constrained. This is because we do not have access to when the last read or write to the page actually occurred; any instruction executed before the privilege-reduction event could have accessed the page.

2.2 Direct Memory Access

Modern hardware systems allow physical devices to write directly to main memory, without involving the processor. This is called *direct memory access* (DMA). DMA eliminates the overhead of the processor copying data from the device to memory.

Replaying DMA presents some difficulties. In DMA, a device acts as another processor with respect to memory transactions. A single processor system with DMA-enabled devices is effectively a multiprocessor system from replay’s perspective. However, unlike peer processors in an SMP system, the devices do not have an MMU that we can use to interpose on accesses[3]. How are we to involve devices in the CREW protocol?

The key observation is that DMA devices are not generally self-motivated peers. They only write to memory in response to a request from a cpu. Requests typically follow a *transaction* model, where a cpu will specify an operation and an area of memory. The device will access the memory during the operation, and inform the cpu when the operation is completed. After the transaction is finished, the device will not access to the memory again. While this transaction is taking place, it is generally not correct for the cpu to access the memory assigned to the device to do DMA.

If the device follows this type of transaction model, where the device will only access memory between certain well-defined boundaries, and the cpu does not need to access memory to the device until a transaction is completed, and if the hypervisor can interpose and understand the commands from the guest to the device and the device’s responses, we can model the device as a *non-preemptible actor* in the CREW protocol. A non-preemptible actor does explicit acquire and release of pages before and after a transaction, rather than acquiring them on demand and having them preempted, as preemptible actors such as virtual cpus do. When a cpu issues a DMA command to the device, the hypervisor informs the CREW protocol, which acquires the appropriate privileges on behalf of the device (either concurrent-read or exclusive-write, depending on the transaction). When the device informs the cpu that the transaction is done, the hypervisor informs the CREW protocol, which will release access on behalf of the device.

If any virtual cpu tries to access a page in a way that is incompatible with the CREW privileges of some device on a system, the CREW system must block its execution until the device has finished the transaction associated with that page. In this way, we do not need to rely on the correctness of kernels or device drivers inside the virtual machine; only on the correctness of the hardware.

During replay, the constraint replay system must ensure that the DMA is replayed at the proper time with respect to the other processors. If we do not use the device during replay, we must log the data from the DMA during logging in order to replay it from the log during replay; otherwise, we must ensure that the device does the DMA properly.

[3] Some new systems include an IO-MMU, for controlling DMA access to memory. However, these systems are designed to prevent buggy drivers and devices from corrupting system state, and do not necessarily provide ways to continue an interrupted operation after a fault. Re-executing a faulting operation is fundamental to our technique.

> “we use hardware page protections, enforced by the memory management unit (MMU), which will check each read and write as the instruction executes, and cause a fault to the hypervisor on any violation.”

Dunlap et al: **Execution Replay for Multiprocessor Virtual Machines**

VEE '08

If b and c are unrelated, we say that the constraints above are *over-constrained*, because they cause the replay system to run more strictly than necessary: either P_2 must wait until P_1 reaches b (although the data was ready at a), or P_2 stops and waits at c (although it is not necessary to stop and wait until d), or both. Over-constraining reduces the potential parallelism during a replay run, but can be taken advantage of to reduce the number of constraints or simplify logging.

Suppose instead that a and d are writes to one area of memory, and b and c were writes to a second area of memory. In this case, $b \rightarrow c$ would be a necessary constraint. However, the constraint $a \rightarrow d$ would be *redundant*, because the ordering $a \rightarrow d$ it is implied by the constraint $b \rightarrow c$. Removing redundant constraints can decrease the log size.

A logging system for a shared memory system must generate a set of constraints that will satisfy the ordering requirement, but is free to choose any set of constraints that will meet that ordering.

To detect which memory operations need to be ordered, we implement a *concurrent-read, exclusive-write* (CREW) protocol between *virtual cpus* in a multiprocessor virtual machine. This technique for detecting constraints was first introduced by [9]. The CREW protocol stipulates that each shared object may be in one of the following two states:

- *concurrent-read*: All cpus have read permission, but none have write permission.
- *exclusive-write*: One cpu (called the *owner*) has both read and write permission; all virtual cpus have no permission.

Each read or write operation to shared memory is checked for access before executing. If a virtual cpu attempts a memory operation for which it has insufficient access, the CREW system must make requests to the other processors to decrease their permissions, so that it may increase its own. We call these increases and decreases in permissions *CREW events*.

The CREW protocol has the following property: if two memory instructions on different processors access the same page, and one of them is a write, there will be a CREW event between the instructions on each processor. This corresponds precisely with the ordering requirement. We can take advantage of this property to detect potential races and generate constraints sufficient to replay the order of accesses for a given execution.

In order to check for access of shared memory reads and writes, we use hardware page protections, available on all modern desktop processors. Hardware page protections are enforced by the *memory management unit*(MMU), which will check each read and write as the instruction executes, and cause a fault to the hypervisor on any violation. Because the checks are done in hardware, the common case is very fast. It also allows us to interpose on reads and writes without modifying the software running on the guest.

Generating constraints from CREW events is straightforward. Each CREW fault will cause two CREW events: a privilege increase on one processor, and a privilege decrease on another processor. If a is the point of privilege decrease, and b is the point of privilege increase, then the constraint $a \rightarrow b$ will be sufficient to order any reads and writes associated with this CREW event.

To see why this is so, consider a particular interaction between two processors, P_1 and P_2. Suppose that instruction b at P_2 writes to a page which is in concurrent-read mode. This instruction will cause a fault into the CREW subsystem. The CREW system will then reduce the privileges of P_1, and increase the privileges of P_2. Let us call the instruction this privilege-decrease happens instruction a. Processor P_1 has had read permission from the time it received permission through point a. Any instruction during that time may have read the page that b is about to write. We cannot tell when the last access was using page protections, but we know that it will be before a by program order, so constraining $a \rightarrow b$ will give us the ordering we need.

Constraining on privilege-reduction events rather than on the last read does mean that our replay will be over-constrained. This is because we do not have access to when the last read or write to the page actually occurred; any instruction executed before the privilege-reduction event could have accessed the page.

2.2 Direct Memory Access

Modern hardware systems allow physical devices to write directly to main memory, without involving the processor. This is called *direct memory access* (DMA). DMA eliminates the overhead of the processor copying data from the device to memory.

Replaying DMA presents some difficulties. In DMA, a device acts as another processor with respect to memory transactions. A single processor system with DMA-enabled devices is effectively a multiprocessor system from replay's perspective. However, unlike peer processors in an SMP system, the devices do not have an MMU that we can use to interpose on accesses[3]. How are we to involve devices in the CREW protocol?

The key observation is that DMA devices are not generally self-motivated peers. They only write to memory in response to a request from a cpu. Requests typically follow a *transaction* model, where a cpu will specify an operation and an area of memory. The device will access the memory during the operation, and inform the cpu when the operation is completed. After the transaction is finished, the device will not access to the memory again. While this transaction is taking place, it is generally not correct for the cpu to access the memory assigned to the device to do DMA.

If the device follows this type of transaction model, where the device will only access memory between certain well-defined boundaries, and the cpu does not need to access memory to the device until a transaction is completed, and if the hypervisor can interpose and understand the commands from the guest to the device and the device's responses, we can model the device as a *non-preemptible actor* in the CREW protocol. A non-preemptible actor does explicit acquire and release of pages before and after a transaction, rather than acquiring them on demand and having them preempted, as preemptible actors such as virtual cpus do. When a cpu issues a DMA command to the device, the hypervisor informs the CREW protocol, which acquires the appropriate privileges on behalf of the device (either concurrent-read or exclusive-write, depending on the transaction). When the device informs the cpu that the transaction is done, the hypervisor informs the CREW protocol, which will release access on behalf of the device.

If any virtual cpu tries to access a page in a way that is incompatible with the CREW privileges of some device on a system, the CREW system must block its execution until the device has finished the transaction associated with that page. In this way, we do not need to rely on the correctness of kernels or device drivers inside the virtual machine; only on the correctness of the hardware.

During replay, the constraint replay system must ensure that the DMA is replayed at the proper time with respect to the other processors. If we do not use the device during replay, we must log the data from the DMA during logging in order to replay it from the log during replay; otherwise, we must ensure that the device does the DMA properly.

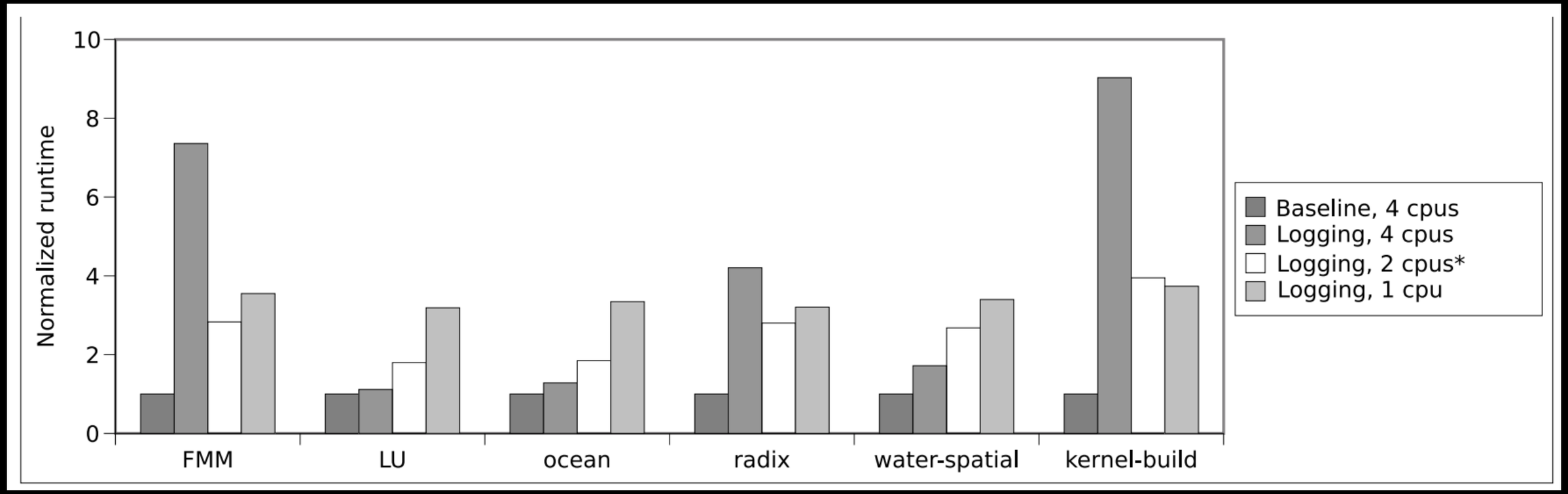
[3] Some new systems include an IO-MMU, for controlling DMA access to memory. However, these systems are designed to prevent buggy drivers and devices from corrupting system state, and do not necessarily provide ways to continue an interrupted operation after a fault. Re-executing a faulting operation is fundamental to our technique.

- **concurrent-read**: All cpus have read permission, but none have write permission.
- **exclusive-write**: One cpu (called the owner) has both read and write permission; all virtual cpus have no permission.

"if two memory instructions on different processors access the same page, and one of them is a write, there will be a CREW event between the instructions on each processor."

CREW limitations

- virtual memory hardware only operates on 4k blocks
- DMA not implemented in the paper
- 10x slowdown in the worst case!



Workload	Logging rate, compressed	Fills 300GB disk in
FMM	83.6 GB/day	3.6 days
LU	11.7 GB/day	25.7 days
ocean	28.1 GB/day	10.7 days
radix	88.7 GB/day	3.4 days
water-spatial	58.5 GB/day	5.1 days
kernel-build	90.0 GB/day	3.3 days

Table 3. Space overhead of logging a four processor guest.

Race detection

race detection techiques

- Lockset algorithm
- Happens-before relation
- Hybrid techniques

```
int var;

void Thread1() {  // Runs in one thread.
  var++;
}
void Thread2() {  // Runs in another thread.
  var++;
}
```

examples from https://github.com/google/sanitizers/wiki/ThreadSanitizerPopularDataRaces

```
// Ref() and Unref() may be called from several threads.
// Last Unref() destroys the object.
class RefCountedObject {
 ...
 public:
  void Ref() {
    ref_++;  // Bug!
  }
  void Unref() {
    if (--ref_ == 0)  // Bug! Need to use atomic decrement!
      delete this;
  }
 private:
  int ref_;
};
```

examples from https://github.com/google/sanitizers/wiki/ThreadSanitizerPopularDataRaces

```
bool done = false;

void Thread1() {
  while (!done) {
    do_something_useful_in_a_loop_1();
  }
  do_thread1_cleanup();
}

void Thread2() {
  do_something_useful_2();
  done = true;
  do_thread2_cleanup();
}
```

examples from https://github.com/google/sanitizers/wiki/ThreadSanitizerPopularDataRaces

```
bool done = false;

void Thread1() {
  while (!done) {
    do_something_useful_in_a_loop_1();
  }
  do_thread1_cleanup();
}

void Thread2() {
  do_something_useful_2();
  done = true;
  do_thread2_cleanup();
}
```

examples from https://github.com/google/sanitizers/wiki/ThreadSanitizerPopularDataRaces

```
bool done = false;

void Thread1() {
  while (!done) {
    do_something_useful_in_a_loop_1();
  }
  do_thread1_cleanup();
}

void Thread2() {
  done = true;
  do_something_useful_2();
  do_thread2_cleanup();
}
```

examples from https://github.com/google/sanitizers/wiki/ThreadSanitizerPopularDataRaces

a concurrent execution. Within any single thread, events are ordered in the order in which they occurred. Between threads, events are ordered according to the properties of the synchronization objects they access. If one thread accesses a synchronization object, and the next access to the object is by a different second if the se

Eraser: A

We continue variable enters ent:

Let *locks_hel*
Let *write_loc*
For each v, i
On each read of v by thread t,
  set $C(v) := C(v) \cap locks\_held(t)$;
  if $C(v) := \{\}$, then issue a warning.
On each write of v by thread t,
  set $C(v) := C(v) \cap write\_locks\_held(t)$;
  if $C(v) = \{\}$, then issue a warning.

That is, locks held purely in read mode are removed from the candidate set when a write occurs, as such locks held by a writer do not protect against a data race between the writer and some other reader thread.

3. IMPLEMENTING ERASER

Eraser is implemented for the Digital Unix operating system on the Alpha processor, using the ATOM [Srivastava and Eustace 1994] binary modification system. Eraser takes an unmodified program binary as input and adds instrumentation to produce a new binary that is functionally identical, but includes calls to the Eraser runtime to implement the Lockset algorithm.

To maintain $C(v)$, Eraser instruments each load and store in the program. To maintain $lock\_held(t)$ for each thread t, Eraser instruments each call to acquire or release a lock, as well as the stubs that manage thread initialization and finalization. To initialize $C(v)$ for dynamically allocated data, Eraser instruments each call to the storage allocator.

Eraser treats each 32-bit word in the heap or global data as a possible shared variable, since on our platform a 32-bit word is the smallest memory-coherent unit. Eraser does not instrument loads and stores whose address mode is indirect off the stack pointer, since these are assumed to be stack references, and shared variables are assumed to be in global locations or in the heap. Eraser will maintain candidate sets for stack locations that are accessed via registers other than the stack pointer, but this is an artifact of the implementation rather than a deliberate plan to support programs that share stack locations between threads.

When a race is reported, Eraser indicates the file and line number at which it was discovered and a backtrace listing of all active stack frames. The report also includes the thread ID, memory address, type of memory

> "Eraser instruments each call to acquire or release a lock"

tual machines, this includes logging virtual interrupts, input from virtual devices such as the virtual keyboard, network, or real-time clock, and the results of non-deterministic instructions such as those that read the processor's time-stamp counter (TSC).

There are two aspects of an event which may be non-deterministic: data, and timing. We call an event which is non-deterministic in data an *input event*. An instruction which reads a processor's TSC is an example of an input event. The result of the read is non-deterministic; but the timing of it is *synchronous* – that is, it always happens at the same point in the instruction stream. To replay these events, the replay system needs to log and replay the data changed by the event.

An event which is non-deterministic in timing is called an *asynchronous* event. A virtual interrupt is an example of an asynchronous event. The state change caused by an interrupt is deterministic (writing certain values on the processor's stack and changing certain registers), but the point in the instruction stream where the interrupt is delivered is non-deterministic.

To replay asynchronous events, an execution replay system needs to be able to identify the exact point in the instruction stream where the event occurred, and replay the event at the same point in the instruction stream during replay. In order to do this, we utilize the hardware branch counter available on several architectures, in conjunction with the instruction address. The observation is that if a given virtual address is executed twice, there must be a branch

2.1 Replaying shared-memory systems

When replaying shared-memory systems, reads from memory by one processor are affected by writes of another processor. Since these reads and writes may happen in any arbitrary interleaving, this introduces fine-grained non-determinism into any shared memory operation.

In order to reconstruct the state of shared memory, each processor must view writes to shared memory by other processors as asynchronous events[1]. We therefore need to preserve the *order* of execution between the processors. We do not need a strict instruction-by-instruction ordering, however. Only reads and writes to shared memory need to be ordered with respect to each other. More specifically, two instructions need to be ordered only if both of the following are true:

- They both access the same memory.
- At least one of them is a write.

This is the *ordering requirement*. Any interleaving of instructions during replay that satisfies the ordering requirement will result in the same execution[2].

We indicate that instruction a is ordered before instruction b by $a \rightarrow b$. This is read, "a happens-before b". In order to enforce an order between two processors, we introduce *constraints* between

requests to the other processors to decrease their permissions, so

the device will only access memory between certain well-defined

> "In order to reconstruct the state of shared memory, each processor must view writes to shared memory by other processors as asynchronous events. We therefore need to preserve the order of execution between the processors."

Savage et al: **Eraser: A Dynamic Data Race Detector for Multithreaded Programs**

SOSP '97

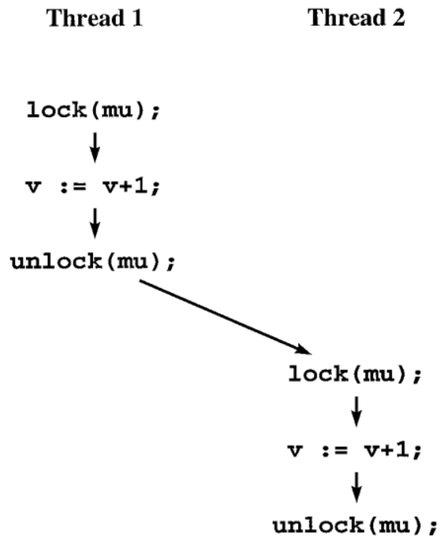


Fig. 1. Lamport's *happens-before* orders events in the same thread in temporal order, and orders events in different threads if the threads are synchronized with one another between the events.

which these two interactions are exchanged in time. For example, Figure 1 shows one possible ordering of two threads executing the same code segment. The three program statements executed by Thread 1 are ordered by *happens-before* because they are executed sequentially in the same thread. The lock of **mu** by Thread 2 is ordered by *happens-before* with the unlock of **mu** by Thread 1 because a lock cannot be acquired before its previous owner has released it. Finally, the three statements executed by Thread 2 are ordered by *happens-before* because they are executed sequentially within that thread.

If two threads both access a shared variable, and the accesses are not ordered by the *happens-before* relation, then in another execution of the program in which the slower thread ran faster and/or the faster thread ran slower, the two accesses could have happened simultaneously; that is, a data race could have occurred, whether or not it actually did occur. All previous dynamic race detection tools that we know of are based on this observation. These race detectors monitor every data reference and synchronization operation and check for conflicting accesses to shared variables that are unrelated by the *happens-before* relation for the particular

Savage et al: **Eraser: A Dynamic Data Race Detector for Multithreaded Programs**

SOSP '97

2. THE LOCKSET ALGORITHM

In this section we describe how the Lockset algorithm detects races. The discussion is at a fairly high level; the techniques used to implement the algorithm efficiently will be described in the following section.

The first and simplest version of the Lockset algorithm enforces the simple locking discipline that every shared variable is protected by some lock, in the sense that the lock is held by any thread whenever it accesses the variable. Eraser checks whether the program respects this discipline by monitoring all reads and writes as the program executes. Since Eraser has no way of knowing which locks are intended to protect which variables, it must infer the protection relation from the execution history.

For each shared variable v, Eraser maintains the set $C(v)$ of candidate locks for v. This set contains those locks that have protected v for the computation so far. That is, a lock l is in $C(v)$ if, in the computation up to that point, every thread that has accessed v was holding l at the moment of the access. When a new variable v is initialized, its candidate set $C(v)$ is considered to hold all possible locks. When the variable is accessed, Eraser updates $C(v)$ with the intersection of $C(v)$ and the set of locks held by the current thread. This process, called lockset refinement, ensures that any lock that consistently protects v is contained in $C(v)$. If some lock l consistently protects v, it will remain in $C(v)$ as $C(v)$ is refined. If $C(v)$ becomes empty this indicates that there is no lock that consistently protects v.

In summary, here is the first version of the Lockset algorithm:

Let *locks_held*(t) be the set of locks held by thread t.
For each v, initialize $C(v)$ to the set of all locks.
On each access to v by thread t,
  set $C(v) := C(v) \cap$ *locks_held*(t);
  if $C(v) = \{\ \}$, then issue a warning.

Figure 3 illustrates how a potential data race is discovered through lockset refinement. The left column contains program statements, executed in order from top to bottom. The right column reflects the set of candidate locks, $C(v)$, after each statement is executed. This example has two locks, so $C(v)$ starts containing both of them. After **v** is accessed while holding **mu1**, $C(v)$ is refined to contain that lock. Later, v is accessed again, with only **mu2** held. The intersection of the singleton sets **{mu1}** and **{mu2}** is the empty set, correctly indicating that no lock protects v.

2.1 Improving the Locking Discipline

The simple locking discipline we have used so far is too strict. There are

Lamport et al: **Time, Locks and the Ordering of Events in a Distributed System**

CACM '78

Operating Systems

R. Stockton Gaines Editor

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

The concept of one event happening before another in a distributed system is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events. The use of the total ordering is illustrated with a method for solving synchronization problems. The algorithm is then specialized for synchronizing physical clocks, and a bound is derived on how far out of synchrony the clocks can become.

Key Words and Phrases: distributed systems, computer networks, clock synchronization, multiprocess systems

CR Categories: 4.32, 5.29

Introduction

The concept of time is fundamental to our way of thinking. It is derived from the more basic concept of the order in which events occur. We say that something happened at 3:15 if it occurred *after* our clock read 3:15 and *before* it read 3:16. The concept of the temporal ordering of events pervades our thinking about systems. For example, in an airline reservation system we specify that a request for a reservation should be granted if it is made *before* the flight is filled. However, we will see that this concept must be carefully reexamined when considering events in a distributed system.

A distributed system consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages. A network of interconnected computers, such as the ARPA net, is a distributed system. A single computer can also be viewed as a distributed system in which the central control unit, the memory units, and the input-output channels are separate processes. A system is distributed if the message transmission delay is not negligible compared to the time between events in a single process.

We will concern ourselves primarily with systems of spatially separated computers. However, many of our remarks will apply more generally. In particular, a multiprocessing system on a single computer involves problems similar to those of a distributed system because of the unpredictable order in which certain events can occur.

In a distributed system, it is sometimes impossible to say that one of two events occurred first. The relation "happened before" is therefore only a partial ordering of the events in the system. We have found that problems often arise because people are not fully aware of this fact and its implications.

In this paper, we discuss the partial ordering defined by the "happened before" relation, and give a distributed algorithm for extending it to a consistent total ordering of all the events. This algorithm can provide a useful mechanism for implementing a distributed system. We illustrate its use with a simple method for solving synchronization problems. Unexpected, anomalous behavior can occur if the ordering obtained by this algorithm differs from that perceived by the user. This can be avoided by introducing real, physical clocks. We describe a simple method for synchronizing these clocks, and derive an upper bound on how far out of synchrony they can drift.

The Partial Ordering

Most people would probably say that an event a happened before an event b if a happened at an earlier time than b. They might justify this definition in terms of physical theories of time. However, if a system is to meet a specification correctly, then that specification must be given in terms of events observable within the system. If the specification is in terms of physical time, then the system must contain real clocks. Even if it does contain real clocks, there is still the problem that such clocks are not perfectly accurate and do not keep precise physical time. We will therefore define the "happened before" relation without using physical clocks.

Lamport et al: **Time, Locks and the Ordering of Events in a Distributed System**

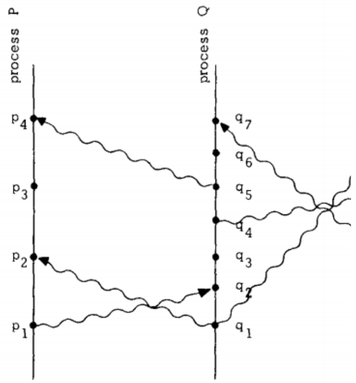
CACM '78

made *before* the flight is filled. However, we will see that this concept must be carefully reexamined when considering events in a distributed system.

This work was supported by the Advanced R Agency of the Department of Defense and Rome A Center. It was monitored by Rome Air Developme contract number F 30602-76-C-0094.

Author's address: Computer Science Laborator tional, 333 Ravenswood Ave., Menlo Park CA 94025.

Fig. 1.



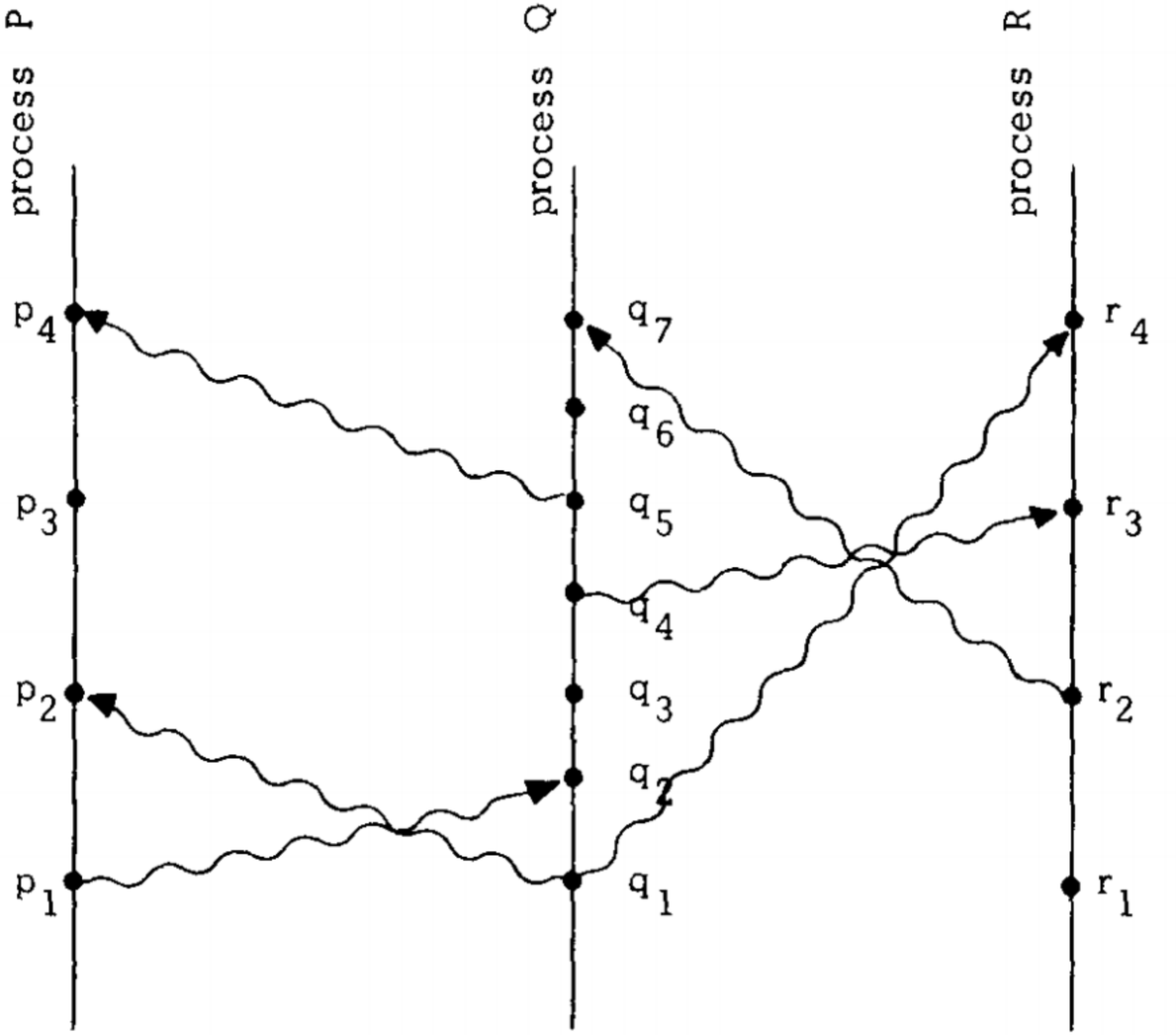
event. We are assuming that the events of a sequence, where *a* occurs before *b* in tl *a* happens before *b*. In other words, a si defined to be a set of events with an ordering. This seems to be what is genera process.[1] It would be trivial to extend ot allow a process to split into distinct subpr will not bother to do so.

We assume that sending or receiving event in a process. We can then define before" relation, denoted by "→", as foll

Definition. The relation "→" on the s a system is the smallest relation satisfyin three conditions: (1) If *a* and *b* are even process, and *a* comes before *b*, then $a \rightarrow$ sending of a message by one process and *b* is the receipt of the same message by another process, then $a \rightarrow b$. (3)

of physical theories of time. However, if a system is to

Fig. 1.



first chapter of [2]. In relativity, the ordering of events is

Dunlap et al: **Execution Replay for Multiprocessor Virtual Machines**

VEE '08

Execution Replay for Multiprocessor Virtual Machines

George W. Dunlap, Dominic G. Lucchetti, Peter M. Chen
Electrical Engineering and Computer Science Dept.
University of Michigan
Ann Arbor, MI 48109-2122
{dunlapg,dlucchet,pmchen}@umich.edu

Michael A. Fetterman
University of Cambridge Computer Laboratory
15 JJ Thompson Avenue, Cambridge, UK, CB3 0FD
Michael.Fetterman@cl.cam.ac.uk

Abstract

Execution replay of virtual machines is a technique which has many important applications, including debugging, fault-tolerance, and security. Execution replay for single processor virtual machines is well-understood, and available commercially. With the advancement of multi-core architectures, however, multiprocessor virtual machines are becoming more important. Our system, SMP-ReVirt, is the first system to log and replay a multiprocessor virtual machine on commodity hardware. We use hardware page protection to detect and accurately replay sharing between virtual cpus of a multi-cpu virtual machine, allowing us to replay the entire operating system and all applications. We have tested our system on a variety of workloads, and find that although sharing under SMP-ReVirt is expensive, for many workloads and applications, including debugging, the overhead is acceptable.

Categories and Subject Descriptors C.4 [*Computer Systems Organization*]: Performance of Systems — Measurement Techniques; D.4.1 [*Operating Systems*]: Process Management — multiprocessing

General Terms Design, Measurement, Performance, Reliability, Security

Keywords ReVirt, execution replay, multithreading, determinism, race recording, multiprocessors, virtual machines, Xen, direct memory access, SPLASH, page protections

1. Introduction

Execution replay gives the ability to reconstruct the past execution of a system. In conjunction with a checkpoint of the system state, it gives the ability to reconstruct the entire state at any point in time over the replay interval. This ability is useful for several different applications. For debugging, it allows a programmer to inspect the execution and state of a particular run of a system, even in the face of non-determinism[9, 14, 5, 8]. For security, it allows a system administrator to go back and inspect the entire state of the system before, during, and after an attack, allowing the system administrator to determine how the attack took place and observe the attacker's activities[6]. For fault tolerance, execution replay allows the state of a system just before a crash to be recovered without the need for frequent checkpoints[7, 4, 11]. Recent work has also used execution replay to efficiently collect and store software architectural traces[19].

A simple way to apply execution replay to a wide range of software is to implement execution replay for virtual machines[4]. Running software in a virtual machine capable of being replayed allows a user to take advantage of execution replay without needing to modify software running inside the virtual machine. It also has the advantage of being able to use execution replay on an operating system kernel.

In order to implement execution replay in a virtual machine, any non-deterministic event that affects the virtual machine's state must be recorded. This state includes all memory allocated to the virtual machine, the processor registers, and the disk. For a single processor system, non-deterministic events include any external input (such as keyboard, mouse, or network), as well as the timing of non-deterministic events like interrupts. The techniques for replaying single processor systems are well understood, and are even available commercially[19].

With the increasing prevalence of multi-core processors, execution replay on multiprocessor systems has become more important. Implementing replay for multiprocessor systems is much more challenging than single processor systems, however. Because writes on one processor can affect reads on another processor, the results of memory races must be recorded and replayed. Existing solutions require modification to software, or massive modifications to hardware.

We have built a system, SMP-ReVirt, which is the first system to log and replay multiprocessor virtual machines on commodity hardware. In order to detect and replay the results of memory races, we use hardware page protections, available on all modern desktop and server processors. This technique allows us to log and replay unmodified multiprocessor systems, including multiprocessor kernels running inside of a virtual machine.

Logging makes sharing more expensive, but the end-to-end impact on performance varies widely depending on the workload. For some applications it is prohibitively expensive, while for others there is little impact.

This paper explores execution replay for multiprocessor virtual machines. Section 2 introduces the basic concepts, terms, and requirements of execution replay for single processor virtual machines. It then discusses the complications that shared-memory systems introduce, and describes techniques to address them. Section 3 describes the research prototype we built using the Xen hypervisor, describing the implementation of the general principles in more detail, and describing some of the technical issues involved. In Section 4, we evaluate our research prototype, investigating the

VEE'08, March 5–7, 2008, Seattle, Washington, USA.

Dunlap et al: **Execution Replay for Multiprocessor Virtual Machines**

VEE '08

source of overhead and sharing. Finally, Section 5 discusses related work.

2. Execution Replay

Logging and replay is widely used for recovering state. The basic concept is straightforward: start from a checkpoint of a prior state, then roll forward, replaying events from the log to reach the desired state. The type of system being recovered determines the type of information that needs to be logged: database logs contain transaction records, file system logs contain file system data, and so on. Replaying a virtual machine requires logging the non-deterministic events that affect the virtual machine's computation. These log records guide the virtual machine as it re-executes (rolls forward) from a checkpoint. Most events are deterministic (e.g. arithmetic, memory, branch instructions) and do not need to be logged; the virtual machine will re-execute these events in the same way during replay as it did during the original execution.

In order to replay an execution, we simply log and replay any non-deterministic event that affects the state of the system. For virtual machines, this includes logging virtual interrupts, input from virtual devices such as the virtual keyboard, network, or real-time clock, and the results of non-deterministic instructions such as those that read the processor's time-stamp counter (TSC).

There are two aspects of an event which may be non-deterministic: data, and timing. We call an event which is non-deterministic in data an *input event*. An instruction which reads a processor's TSC is an example of an input event. The result of the read is non-deterministic; but the timing of it is *synchronous* – that is, it always happens at the same point in the instruction stream. To replay these events, the replay system needs to log and replay the data changed by the event.

An event which is non-deterministic in timing is called an *asynchronous* event. A virtual interrupt is an example of an asynchronous event. The state change caused by an interrupt is deterministic (writing certain values on the processor's stack and changing certain registers), but the point in the instruction stream where the interrupt is delivered is non-deterministic.

To replay asynchronous events, an execution replay system needs to be able to identify the exact point in the instruction stream where the event occurred, and replay the event at the same point in the instruction stream during replay. In order to do this, we utilize the hardware branch counter available on several architectures, in conjunction with the instruction address. The observation is that if a given virtual address is executed twice, there must be a branch between them. Using branch counters allows us to identify a particular instruction in the instruction stream at which the asynchronous event occurred, so that we can re-deliver the event at the same point during replay.

Note that an event may be both asynchronous and an input event. An example of such an event is DMA from a virtual device, where both the timing of the DMA, and the data written by the DMA, must be logged and replayed for the system work correctly.

Input from devices, such as keyboard and network, must be logged and replayed; however, output, such as writes to a console or sending network packets, do not affect state and do not need to be logged or replayed. The data sent will be re-generated by the replaying system. This data may be discarded without affecting the reconstruction of the state of the virtual machine. However, it is frequently useful to involve those devices in replay.

Other devices, such as the disk, allow us a choice. We could simply log all reads from the disk; but this typically generates a prohibitive amount of data, even for a moderately short run. Instead, we can avoid logging input from the disk by including it in the replaying system. If we checkpoint and restore the disk along with the rest of the state of the system, writes to the disk will be re-generated, which causes reads to return the same data as during logging. Thus we can make reads from disk deterministic without logging them.

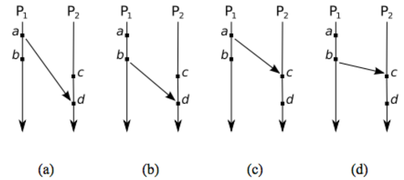


Figure 1. Constraints sufficient to guarantee the order $a \rightarrow d$

2.1 Replaying shared-memory systems

When replaying shared-memory systems, reads from memory by one processor are affected by writes of another processor. Since these reads and writes may happen in any arbitrary interleaving, this introduces fine-grained non-determinism into any shared memory operation.

In order to reconstruct the state of shared memory, each processor must view writes to shared memory by other processors as asynchronous events[1]. We therefore need to preserve the *order* of execution between the processors. We do not need a strict instruction-by-instruction ordering, however. Only reads and writes to shared memory need to be ordered with respect to each other. More specifically, two instructions need to be ordered only if both of the following are true:

- They both access the same memory.
- At least one of them is a write.

This is the *ordering requirement*. Any interleaving of instructions during replay that satisfies the ordering requirement will result in the same execution[2].

We indicate that instruction a is ordered before instruction b by $a \rightarrow b$. This is read, "a happens-before b". In order to enforce an order between two processors, we introduce *constraints* between instructions. A constraint $a \rightarrow b$ indicates that the replay system will ensure that b does not execute until a has executed.

Two points on the instruction stream may be ordered even if there is no direct constraint from one to the other. Within a single processor, there is an implicit ordering, based on the order the instructions were executed. Furthermore, ordering is transitive: $a \rightarrow b$ and $b \rightarrow c$ implies $a \rightarrow c$.

Consider Figure 1. Suppose that a and d are writes to the same memory, but that b and c are unrelated—$a \rightarrow d$ is the only ordering necessary. One constraint sufficient to guarantee the ordering is $a \rightarrow d$. But any of the following constraints would imply the order $a \rightarrow b$ as well:

- $b \rightarrow d$ (because $a \rightarrow b$ by program order)
- $a \rightarrow c$ (because $c \rightarrow d$ by program order)
- $b \rightarrow c$ (because $a \rightarrow b$ and $c \rightarrow d$ by program order)

[1] We can instead view reads from shared memory as synchronous data input events. This is the method taken by BugNet[12], discussed in Section 5.

[2] See [10] for a more complete exploration of the concept of order and state in a distributed system.

If b and c are unrelated, we say that the constraints above are

be before a by program order, so constraining $a \rightarrow b$ will give us

"In order to reconstruct the state of shared memory, each processor must view writes to shared memory by other processors as asynchronous events. We therefore need to preserve the order of execution between the processors."

"Only reads and writes to shared memory need to be ordered with respect to each other"

Savage et al: **Eraser: A Dynamic Data Race Detector for Multithreaded Programs**

SOSP '97

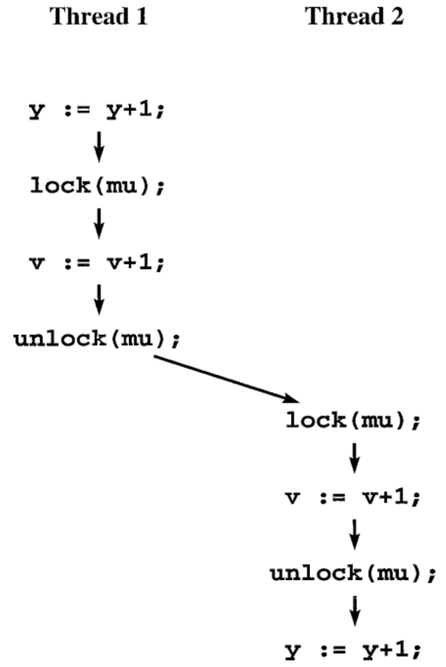


Fig. 2. The program allows a data race on **y**, but the error is not detected by *happens-before* in this execution interleaving.

Figure 2 shows a simple example where the *happens-before* approach can miss a data race. While there is a potential data race on the unprotected accesses to **y**, it will not be detected in the execution shown in the figure, because Thread 1 holds the lock before Thread 2, and so the accesses to **y** are ordered in this interleaving by *happens-before*. A tool based on *happens-before* would detect the error only if the scheduler produced an interleaving in which the fragment of code for Thread 2 occurred before the fragment of code for Thread 1. Thus, to be effective, a race detector based on *happens-before* needs a large number of test cases to test many possible interleavings. In contrast, the programming error in Figure 2 will be detected by Eraser with any test case that exercises the two code paths, because the paths violate the locking discipline for **y** regardless of the interleaving produced by the scheduler. While Eraser is a testing tool and therefore cannot guarantee that a program is free from races, it can detect more

O'Callahan & Choi: **Hybrid Dynamic Data Race Detection**

PPoPP '03

- $\mathcal{T}$: a set of *threads*. In Java, a thread corresponds to an object of class `Thread`.
- $\mathcal{G}$: a set of *message IDs*. In Java, an example of a message is an object that is synchronized on using `wait()` and `notify()`.
- $\mathcal{A} = \{\text{READ, WRITE}\}$: the two possible *access types* for a memory access.

Program execution generates the following kinds of events:

- *Memory access* events of the form $\text{MEM}(m, a, t)$ where $m \in \mathcal{M}$, $a \in \mathcal{A}$ and $t \in \mathcal{T}$. These indicate that thread t performed an access of type a to location m. In Java these correspond to reading and assigning the values of static and non-static fields, and reading and assigning array elements.
- *Lock acquisition* events of the form $\text{ACQ}(l, t)$ where $l \in \mathcal{L}$ and $t \in \mathcal{T}$. These indicate that thread t acquired lock l. (Java locks are reentrant; we only generate ACQ when t did not already hold the lock.) In Java these correspond to entering a `synchronized` method or block where t did not already hold the lock.
- *Lock release* events of the form $\text{REL}(l, t)$ where $l \in \mathcal{L}$ and $t \in \mathcal{T}$. These indicate that thread t released lock l and no longer holds the lock. In Java these correspond to leaving a `synchronized` method or block where t usage count on the lock decreases to zero.
- *Thread message send* events of the form $\text{SND}(g, t)$ where $t \in \mathcal{T}$ and $g \in \mathcal{G}$. These indicate that thread t is sending a message g to some receiving thread.
- *Thread message receive* events of the form $\text{RCV}(g, t)$ where $t \in \mathcal{T}$ and $g \in \mathcal{G}$. These indicate that a thread t has received a message g from some sending thread and may now be unblocked if it was blocked before.

Thread message events are only observed by the happens-before detector, discussed in Section 3.

To simplify the presentation, we assume the abstract machine is sequential. At each step, it chooses a single thread to run, and executes that thread for some quantum, possibly generating one or more events. Thus events are observed by our detector in a sequence which depends on the thread schedule. Our implementation uses locks inside the detector to map Java thread execution into this sequential abstraction.

2.2 Accumulating Locksets

Before performing lockset-based detection, we must compute the set of locks held by a thread at any given time.

Given an access sequence $\langle e_i \rangle$, we compute the locks before step i by a thread t, $L_i(t)$, as

$$L_i(t) = \{\, l \mid \exists a.a < i \wedge e_a = \text{ACQ}(l, t) \wedge (\nexists r.a < r < i \wedge e_r = \text{REL}(l, t))\}$$

The "current lockset" for each thread, $L_i(t)$ for each live thread t, can be efficiently maintained online as acquisition and release events are received.

2.3 The Lockset Hypothesis

Lockset-based detection relies on the following hypothesis: *Whenever two different threads access a shared memory location, and one of the accesses is a write, the two accesses are performed holding some common lock.* The postulated lock ensures mutual exclusion for the two accesses to the shared location. A potential race is deemed to have occurred whenever this hypothesis is violated. Formally, given an input sequence $\langle e_i \rangle$,

$$\begin{aligned}&\text{IsPotentialLocksetRace}(i, j) = \\ &\quad e_i = \text{MEM}(m_i, a_i, t_i) \wedge e_j = \text{MEM}(m_j, a_j, t_j) \\ &\quad \wedge t_i \neq t_j \wedge\ m_i = m_j \wedge (a_i = \text{WRITE} \vee a_j = \text{WRITE}) \\ &\quad \wedge L_i(t_i) \cap L_j(t_j) = \emptyset\end{aligned}$$

For example, in Figure 1, the statement "`childThread.interrupt()`" generates a memory access with location `main.childThread`, type READ, thread MAIN, and lockset {`main`}. The statement "`main.childThread`" generates a memory access on `main.childThread` with type WRITE, thread CHILD and lockset $\emptyset$. Therefore IsPotentialLocksetRace will be true for these two events.

2.4 Lockset-Based Detection

Because the number of races is potentially quadratic in the number of memory accesses, we cannot report all races, nor would that be useful in practice. Instead our tool reports one race for each memory location m on which at least one potential race is detected. This simplification creates opportunities for many optimizations.

To check IsPotentialLocksetRace for all access to a given memory location m, it suffices to store a set of (a, t, L) tuples with the access type, thread, and current lockset for each access to m. Since we only need to detect one race, multiple accesses with identical (a, t, L) tuples are redundant and only one tuple need be recorded. Therefore our basic detection algorithm processes a $\text{MEM}(m, a, t)$ event by first checking to see whether the (a, t, L) tuple is already present for m. If it is already present, the new access is ignored. Otherwise if (a, t, L) forms a potential race with any prior tuple (a_p, t_p, L_p) according to IsPotentialLocksetRace, a race is reported and we stop detecting races on m. Otherwise we add (a, t, L) to the tuple set for m.

In practice we perform many optimizations to improve this algorithm. The space requirements of the tuple set, and the cost of detecting duplicate and racing tuples, can be significantly reduced by carefully choosing the representation of the tuple set, but for the sake of brevity this paper does not describe those choices. Other optimizations are described below.

3. HAPPENS-BEFORE RACE DETECTION

Unfortunately violations of the lockset hypothesis are not always programming errors. Programmers can and do write safe multi-threaded code which mutates shared data without specific locks protecting the data. One common example is programs which use *channels* to pass objects between threads in the style of CSP [15]. In such programs thread synchronization and mutual exclusion are accomplished by explicit signaling between threads.

Figure 2 shows an example of object recycling, a common technique for reducing object allocation and initialization costs in large programs. Object recycling often leads false positives in a lockset-based detector. The problem is that thread A and thread B both access `myBig` without holding locks, and thus the accesses are reported as races by the lockset-based detector. However races are in fact prevented because exclusive ownership of the BigObject is

"Unfortunately violations of the lockset hypothesis are not always programming errors. One common example is programs which use channels to pass objects between threads in the style of CSP [15].
In such programs thread synchronization and mutual exclusion are accomplished by explicit signaling between threads."

- Lockset - ensure >1 lock always protects shared me
  - + Doesn’t need to observe a bug happening to find a race
  - - False positives if different synchronization techniques used

- Happens-Before - shared accesses must be separated by a sync
  - + Causality def’n doesn’t require locks: fewer false positives
  - - Observing races depends on execution order
  - - Slower to run in practice than Lockset

Serebryany & Iskhodzhanov: **ThreadSanitizer: data race detection in practice** WBIA '12

ThreadSanitizer – data race detection in practice

Konstantin Serebryany
OOO Google
7 Balchug st.
Moscow, 115035, Russia
kcc@google.com

Timur Iskhodzhanov
MIPT
9 Institutskii per.
Dolgoprudny, 141700, Russia
timur.iskhodzhanov@phystech.edu

ABSTRACT

Data races are a particularly unpleasant kind of threading bugs. They are hard to find and reproduce – you may not observe a bug during the entire testing cycle and will only see it in production as rare unexplainable failures. This paper presents ThreadSanitizer – a dynamic detector of data races. We describe the hybrid algorithm (based on happens-before and locksets) used in the detector. We introduce what we call dynamic annotations – a sort of race detection API that allows a user to inform the detector about any tricky synchronization in the user program. Various practical aspects of using ThreadSanitizer for testing multi-threaded C++ code at Google are also discussed.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging — *Testing tools.*

General Terms

Algorithms, Testing, Reliability.

Keywords

Concurrency Bugs, Dynamic Data Race Detection, Valgrind.

1. INTRODUCTION

A data race is a situation when two threads concurrently access a shared memory location and at least one of the accesses is a write.

Such bugs are often difficult to find because they happen only under very specific circumstances which are hard to reproduce. In other words, a successful pass of all tests doesn't guarantee the absence of data races. Since races can result in data corruption or segmentation fault, it is important to have tools for finding existing data races and for catching new ones as soon as they appear in the source code.

WBIA '09, Dec 12, New York City, NY

The problem of precise race detection is known to be NP-hard (see [20]). However, it is possible to create tools for finding data races with acceptable precision (such tools will miss some races and/or report false warnings).

Virtually every C++ application developed at Google is multithreaded. Most of the code is covered with tests, ranging from tiny unit tests to huge integration and regression tests. However, our codebase had never been studied using a data race detector. Our main task was to implement and deploy a continuous process for finding data races.

2. RELATED WORK

There are a number of approaches to data race detection. The three basic types of detection techniques are: static, on-the-fly and postmortem. On-the-fly and postmortem techniques are often referred to as dynamic.

Static data race detectors analyze the source code of a program (e.g. [11]). It seems unlikely that static detectors will work effectively in our environment: Google's code is large and complex enough that it would be expensive to add the annotations required by a typical static detector.

Dynamic data race detectors analyze the trace of a particular program execution. On-the-fly race detectors process the program's events in parallel with the execution [14, 22]. The postmortem technique consists in writing such events into a temporary file and then analyzing this file after the actual program execution [18].

Most dynamic data race detection tools are based on one of the following algorithms: happens-before, lockset or both (the hybrid type). A detailed description of these algorithms is given in [21]. Each of these algorithms can be used in the on-the-fly and postmortem analysis.

3. HISTORY OF THE PROJECT

Late in 2007 we tried several publicly available race detectors, but all of them failed to work properly "out of the box". The best of these tools was Helgrind 3.3 [8] which had a hybrid algorithm. But even Helgrind had too many false positives and missed many real races. Early in 2008 we modified the Helgrind's hybrid algorithm and also introduced an optional pure happens-before mode. The happens-before mode had fewer false positives but missed even more data races than the initial hybrid algorithm. Also, we introduced dynamic annotations (section 5) which helped eliminate false positive reports even in the hybrid mode.

Still, Helgrind did not work for us as effectively as we would like it to — it was still too slow, missed too many

a hybrid approach

- Full happens-before tracking for synchronize operations
- Lockset approach to determine whether causally-related locksets are non-empty

Serebryany & Iskhodzhanov: **ThreadSanitizer: data race detection in practice** WBIA '12

	app		base		ipc		net		unit	
native	3s	172M	77s	1811M	5s	325M	50s	808M	43s	914M
Memcheck-no-hist	6.7x	2.0x	1.7x	1.1x	5.2x	1.1x	3.0x	1.6x	14.8x	1.7x
Memcheck	10.5x	2.6x	2.2x	1.1x	8.2x	1.2x	5.1x	2.3x	29.7x	1.9x
Helgrind-no-hist	13.9x	2.7x	1.8x	1.8x	5.4x	1.5x	4.5x	2.2x	48.7x	3.4x
Helgrind	14.9x	3.8x	1.7x	1.9x	6.7x	1.7x	11.9x	2.5x	62.3x	3.8x
TS-fast-no-hist	6.2x	4.2x	2.2x	1.2x	11.1x	1.8x	3.9x	1.7x	19.2x	2.2x
TS-fast	7.9x	7.6x	2.4x	1.5x	12.0x	3.6x	4.7x	2.4x	21.6x	2.8x
TS-full-no-hist	8.4x	4.2x	2.4x	1.2x	11.3x	1.8x	4.7x	1.6x	22.3x	2.3x
TS-full	13.8x	7.4x	2.8x	1.5x	11.9x	3.6x	6.3x	2.3x	28.6x	2.5x
TS-phb-no-hist	8.3x	4.2x	2.8x	1.2x	11.2x	1.8x	4.7x	1.8x	23.0x	6.2x
TS-phb	14.2x	7.4x	2.6x	1.5x	11.8x	3.6x	6.2x	2.3x	28.6x	2.5x

Serebryany & Iskhodzhanov: **ThreadSanitizer: data race detection in practice** WBIA '12

7. RACE DETECTION FOR CHROMIUM

One of the applications we test with ThreadSanitizer is Chromium [1], an open-source browser project.

The code of Chromium browser is covered by a large number of tests including unit tests, integration tests and interactive tests running the real application. All these tests are continuously run on a large number of test machines with different operating systems. Some of these machines run tests under Memcheck (the Valgrind tool which finds memory-related errors, see [8]) and ThreadSanitizer. When a new error (either a test failure or a race report from ThreadSanitizer) is found after a commit to the repository, the committer of the change is notified. These reports are available for other developers and maintainers as well.

We have found and fixed a few dozen data races in Chromium itself, and in some third party components used by this project. You may find all these bugs by searching for `label:ThreadSanitizer` at `www.crbug.com`.

7.1 Top crasher

One of the first data races we found in Chromium happened to be the cause of a serious bug, which had been observed for several months but had not been understood nor fixed[15]. The data race happened on a class called `RefCounted`. The reference counter was incremented and decremented from multiple threads without synchronization. When the race actually occurred (which happened very rarely), the value of the counter became incorrect. This resulted in either a memory leak or in two calls of `delete` on the same memory. In the latter case, the internals of the memory allocator were corrupted and one of the subsequent calls to `malloc` failed with a segmentation fault.

The cause of these failures was not understood for a long time because the failure never happened during debugging, and the failure stack traces were in a different place. ThreadSanitizer found this data race in a single run.

The fix for this data race was simple. Instead of the `RefCounted` class we needed to use `RefCountedThreadSafe`, the class which implements reference counting using atomic instructions.

[15]See the bug entries http://crbug.com/18488 and http://crbug.com/15577 describing the race and the crashes, respectively.

what have we seen

- Time-Traveling Debugging for reconstructing previous program state
- JIT instrumentation for efficient control flow modification
- Shadow memory for efficient memory introspection

the trace is out there

- Novel and experimental hardware
- Query languages and storage for large trace data
- Trace visualization techniques

Thanks

Slides, Citations,
and Acknowledgments:
https://github.com/dijkstracula/QConNYC2016

Nathan Taylor, Fastly
http://nathan.dijkstracula.net
@dijkstracula