beyond breakpoints

A Tour of Dynamic Analysis

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

All right, thanks so much for coming today, everybody. I know it's the last session of the day so I really appreciate you coming. My name is Nathan Taylor - I'm a developer at a software company called Fastly, and I'd like to tell you about some cool techniques in a field called dynamic analysis.

That’s me up there - I’ve been a low-level systems person for the bulk of my career. Right now I <click> work on low-latency web content delivery; in previous jobs I’ve <click> worked on JVM internals and just-in-time compilers, and in <click> grad school I hacked on hypervisors like Xen, but also spent a lot of time thinking about how to build tools to help people better understand how their software behaves at runtime. Such tools would fall under the umbrella term

fastly®

That’s me up there - I’ve been a low-level systems person for the bulk of my career. Right now I <click> work on low-latency web content delivery; in previous jobs I’ve <click> worked on JVM internals and just-in-time compilers, and in <click> grad school I hacked on hypervisors like Xen, but also spent a lot of time thinking about how to build tools to help people better understand how their software behaves at runtime. Such tools would fall under the umbrella term

That’s me up there - I’ve been a low-level systems person for the bulk of my career. Right now I <click> work on low-latency web content delivery; in previous jobs I’ve <click> worked on JVM internals and just-in-time compilers, and in <click> grad school I hacked on hypervisors like Xen, but also spent a lot of time thinking about how to build tools to help people better understand how their software behaves at runtime. Such tools would fall under the umbrella term

That’s me up there - I’ve been a low-level systems person for the bulk of my career. Right now I <click> work on low-latency web content delivery; in previous jobs I’ve <click> worked on JVM internals and just-in-time compilers, and in <click> grad school I hacked on hypervisors like Xen, but also spent a lot of time thinking about how to build tools to help people better understand how their software behaves at runtime. Such tools would fall under the umbrella term

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 runtime state. This instrumentation can be tuned to collect precisely the information needed

dynamic analysis, which Thomas Ball, in his paper “The Concept of Dynamic Analysis”, simply defines as <click> the analysis of the properties of a running program, which are usually derived through program instrumentation.

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

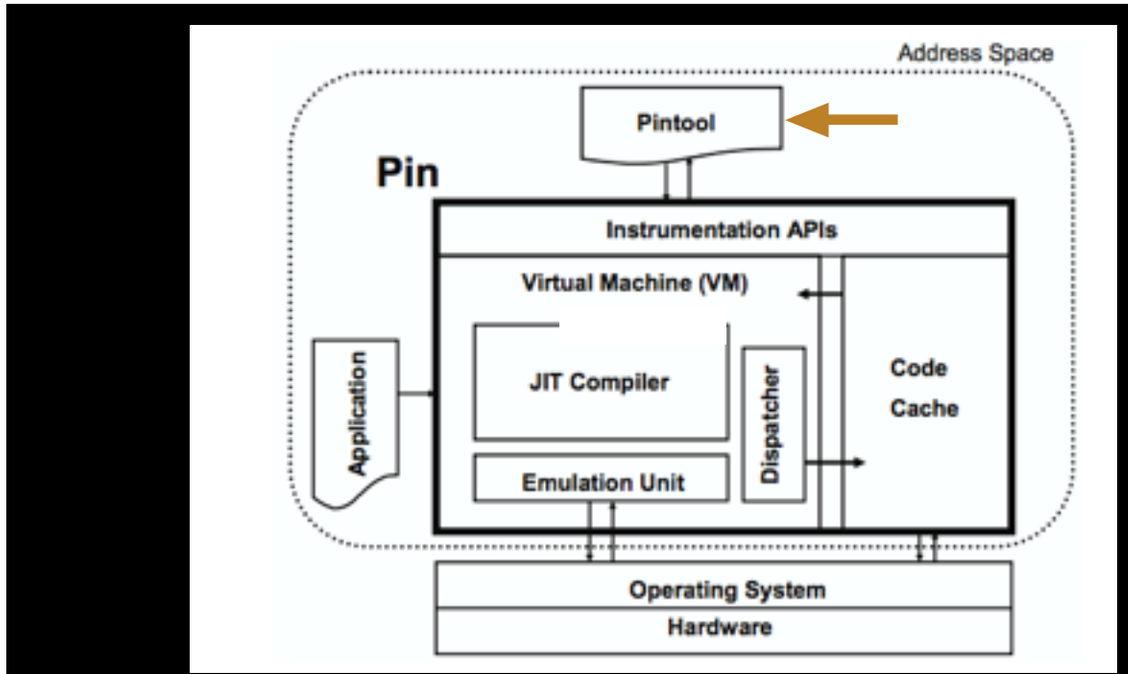
Dynamic analysis is the analysis of the properties of a running program. In contrast to static analysis, which examines a program's text to derive properties that hold for all executions, dynamic analysis derives properties that hold for one or more executions by examination of the running program (usually 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 runtime state. This instrumentation can be tuned to collect precisely the information needed

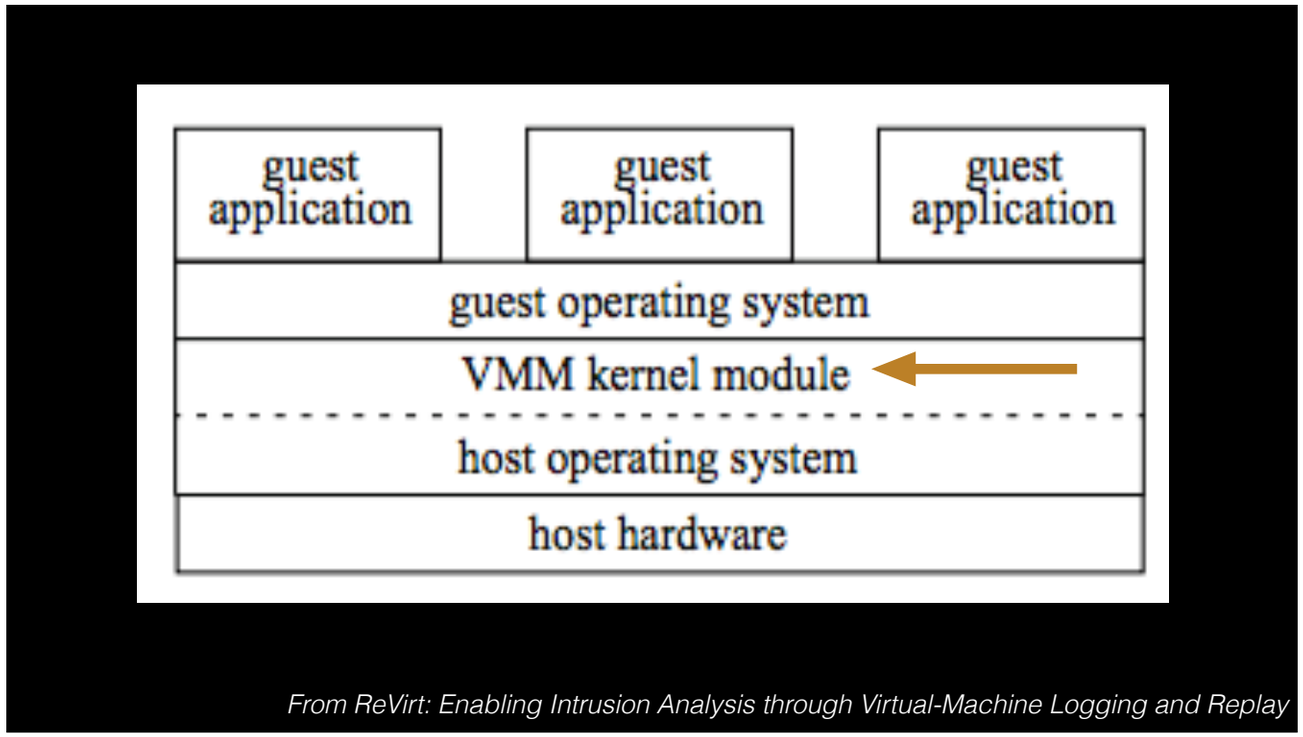
"Dynamic analysis is the analysis of the properties of a running program [...] (usually through program instrumentation)."

dynamic analysis, which Thomas Ball, in his paper "The Concept of Dynamic Analysis", simply defines as <click> the analysis of the properties of a running program, which are usually derived through program instrumentation.



From Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation

So, program instrumentation here can mean two things: if the “system” you’re observing is an application, then the dynamic analysis is running in another program on the same machine. This is sort of like how a debugger attaches to another process.



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

If, on the other hand, the system you want to analyze is an operating system and all the userspace programs running atop it, then the tool will run as part of a hardware emulator like QEMU or a hypervisor like Xen. But, irrespective of whether you're observing a single process or a whole OS, the principle is the same. We sit in between the thing we want to analyze and the rest of the world and observe how it behaves.

classic use-cases

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

There are lots of reasons why dynamic analysis might be useful to developers. As anyone who's used a debugger knows, stepping through a buggy program is invaluable to figure out why something is misbehaving. Similarly, stepping through a program you're unfamiliar with to explore its source code has been a technique I've used, too. And, most interestingly, there's been work in observing malware execute in a cleanroom environment by security researchers to study how malicious software ticks.

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

So, if you're wondering why I think this is interesting and important enough to make a talk about: this is the CS Theory in the Real World track, but there's so much academic work in the dynamic analysis space that I haven't seen applied by practitioners, and I think that's a real shame. Part of my goal here is to evangelize these sorts of ideas.

Second, I think the field is interesting in its own right, particularly because there's a lot of overlap in CS areas that I think are cool - we'll see in this talk that building these sorts of tools pulls in lots of material from everything from operating system to hardware virtualization to language runtimes.

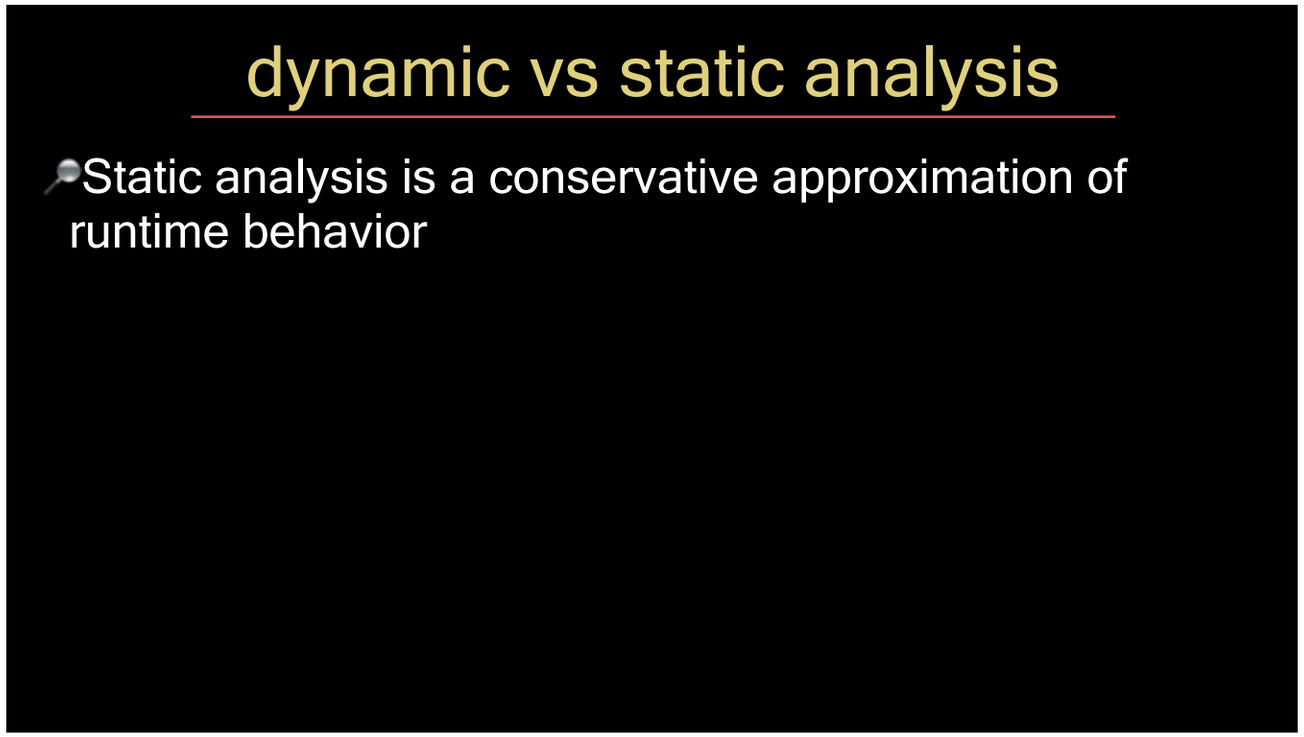
Lastly, there are lots of dynamic analysis techniques that vary in how computationally expensive they are. I think that if people are exposed to a wide range of options they'll be able to make the right tech choice for their particular needs.

dynamic vs static analysis

I think it’s worth taking a moment to contrast dynamic analysis with static techniques like formal verification, which Caitie gave a great talk about earlier today, and language features like type systems. <click>

Well, static analysis is all about proving properties hold for _all executions_, so it’s necessarily a conservative approximation of runtime behaviour. For instance, <click> something that can make Java static analysis hard is that the fundamental unit of compilation is the Class, so reasoning about how that classfile is going to interact with a larger system (particularly with dynamic classloading) is really hard. Under a dynamic analysis tool, you’re watching the complete program execute so this isn’t an issue.

I have a lot of friends who read type theory textbooks for fun and they might say “well just rub some types on your problem”. <click> Sure, I would much rather program in a strongly-typed language than an untyped one but types only get you so far - I think we’ve all been in the situation where our program typechecks but there’s a business logic bug that wasn’t captured in the type system.



I think it’s worth taking a moment to contrast dynamic analysis with static techniques like formal verification, which Caitie gave a great talk about earlier today, and language features like type systems. <click>

Well, static analysis is all about proving properties hold for _all executions_, so it’s necessarily a conservative approximation of runtime behaviour. For instance, <click> something that can make Java static analysis hard is that the fundamental unit of compilation is the Class, so reasoning about how that classfile is going to interact with a larger system (particularly with dynamic classloading) is really hard. Under a dynamic analysis tool, you’re watching the complete program execute so this isn’t an issue.

I have a lot of friends who read type theory textbooks for fun and they might say “well just rub some types on your problem”. <click> Sure, I would much rather program in a strongly-typed language than an untyped one but types only get you so far - I think we’ve all been in the situation where our program typechecks but there’s a business logic bug that wasn’t captured in the type system.

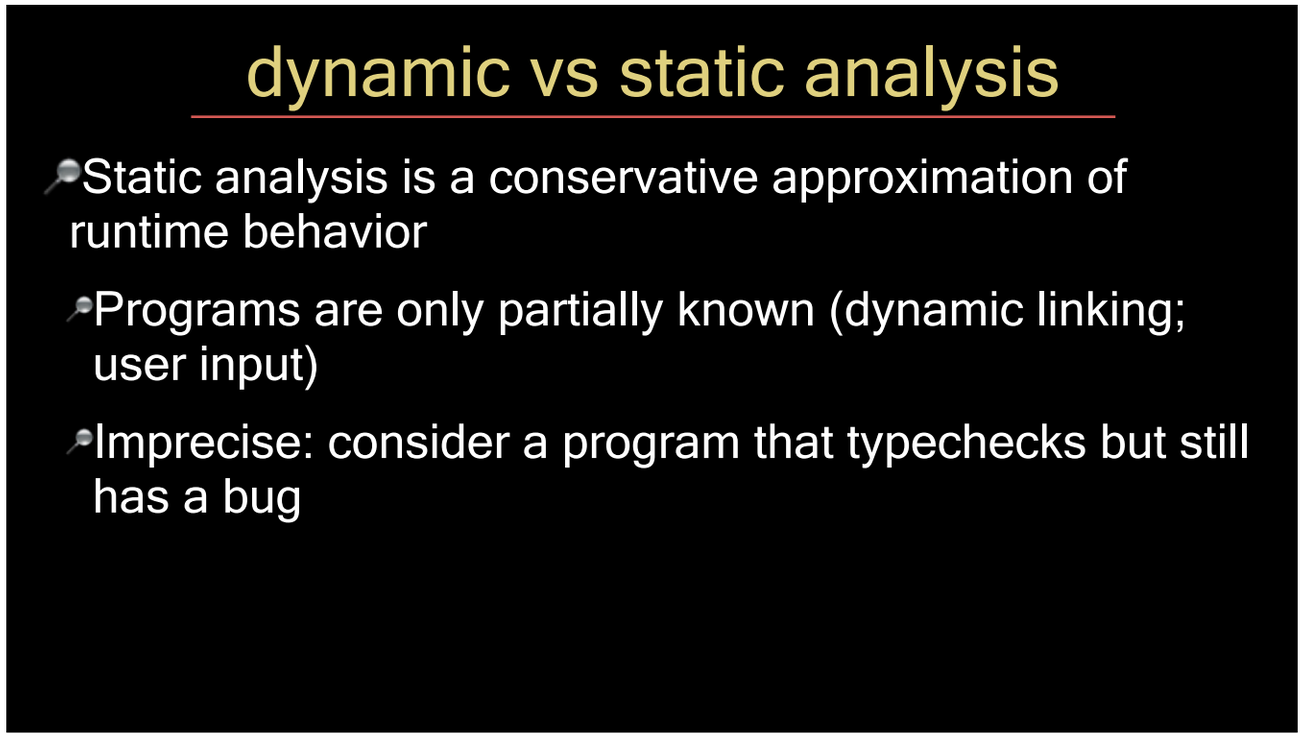
dynamic vs static analysis

- Static analysis is a conservative approximation of runtime behavior
  - Programs are only partially known (dynamic linking; user input)

I think it's worth taking a moment to contrast dynamic analysis with static techniques like formal verification, which Caitie gave a great talk about earlier today, and language features like type systems. <click>

Well, static analysis is all about proving properties hold for _all executions_, so it's necessarily a conservative approximation of runtime behaviour. For instance, <click> something that can make Java static analysis hard is that the fundamental unit of compilation is the Class, so reasoning about how that classfile is going to interact with a larger system (particularly with dynamic classloading) is really hard. Under a dynamic analysis tool, you're watching the complete program execute so this isn't an issue.

I have a lot of friends who read type theory textbooks for fun and they might say "well just rub some types on your problem". <click> Sure, I would much rather program in a strongly-typed language than an untyped one but types only get you so far - I think we've all been in the situation where our program typechecks but there's a business logic bug that wasn't captured in the type system.



I think it's worth taking a moment to contrast dynamic analysis with static techniques like formal verification, which Caitie gave a great talk about earlier today, and language features like type systems. <click>

Well, static analysis is all about proving properties hold for _all executions_, so it's necessarily a conservative approximation of runtime behaviour. For instance, <click> something that can make Java static analysis hard is that the fundamental unit of compilation is the Class, so reasoning about how that classfile is going to interact with a larger system (particularly with dynamic classloading) is really hard. Under a dynamic analysis tool, you're watching the complete program execute so this isn't an issue.

I have a lot of friends who read type theory textbooks for fun and they might say "well just rub some types on your problem". <click> Sure, I would much rather program in a strongly-typed language than an untyped one but types only get you so far - I think we've all been in the situation where our program typechecks but there's a business logic bug that wasn't captured in the type system.

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!

I think it's worth taking a moment to contrast dynamic analysis with static techniques like formal verification, which Caitie gave a great talk about earlier today, and language features like type systems. <click>

Well, static analysis is all about proving properties hold for _all executions_, so it's necessarily a conservative approximation of runtime behaviour. For instance, <click> something that can make Java static analysis hard is that the fundamental unit of compilation is the Class, so reasoning about how that classfile is going to interact with a larger system (particularly with dynamic classloading) is really hard. Under a dynamic analysis tool, you're watching the complete program execute so this isn't an issue.

I have a lot of friends who read type theory textbooks for fun and they might say "well just rub some types on your problem". <click> Sure, I would much rather program in a strongly-typed language than an untyped one but types only get you so far - I think we've all been in the situation where our program typechecks but there's a business logic bug that wasn't captured in the type system.

Dynamic analysis is more about the proof is in the pudding. <click> it can’t prove correctness for all executions of a system, but <click> it can report a particular instance where a failure has occurred.

can’t prove correctness

Dynamic analysis is more about the proof is in the pudding. <click> it can’t prove correctness for all executions of a system, but <click> it can report a particular instance where a failure has occurred.

can’t prove correctness

but can demonstrate failure

Dynamic analysis is more about the proof is in the pudding. <click> it can’t prove correctness for all executions of a system, but <click> it can report a particular instance where a failure has occurred.

today's topics

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

We are going to talk about three major components which I list here - we'll talk about building more sophistiated debuggers and analysis tools, then move on to my favourite topic, concurrency, and then wrap up with talking about everything that we haven't talked about.

for each topic...

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

So for each of these topics, we will dive into open source tooling for each of these problem areas, look at how these tools are actually implemented, and talk about what academia has to offer in terms of research contributions.

So let’s begin with a situation that I think most of us have been in. So you write a great program called Foo.C,

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

you compile and run it, and — uh oh. So this is fine, we'll recompile

```
➜  /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
```

with source symbols and start up our ordinary debugger. <click> and we see, okay, we crashed because we tried to assign a number to an invalid pointer, and

```
➜  /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;
```

and we see, okay, we crashed because we tried to dereference a pointer, and

```
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)
```

with a bit more digging we have a stack trace, and we see that the pointer value that we tried to dereference is the value…7. Well, now what?

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


The Night Watch

JAMES MICKENS

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 skeletons coming home to roast, and fast zombies, and slow zombies, and the polite zombies who say "no" and "excuse" but then try to use 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 ragtag 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. (Seriously, 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 clear goldfish. The systems programmer has traced a network problem across eight machines, three time zones, and a brief diversion into Amish counting, where the problem was transmitted to 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 remnant in the aftermath that says "DOES THIS WORK LOL," and the systems programmer has wept instead of LOLed, and the systems programmer has rolled/tied 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.

Of course, <click> as the great James Mickens tells us, 7 is very far from being a valid memory address. This smells like a bug that scribbled over the pointer with some garbage bytes, but

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

"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. "

Of course, <click> as the great James Mickens tells us, 7 is very far from being a valid memory address. This smells like a bug that scribbled over the pointer with some garbage bytes, but

```
(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)
```

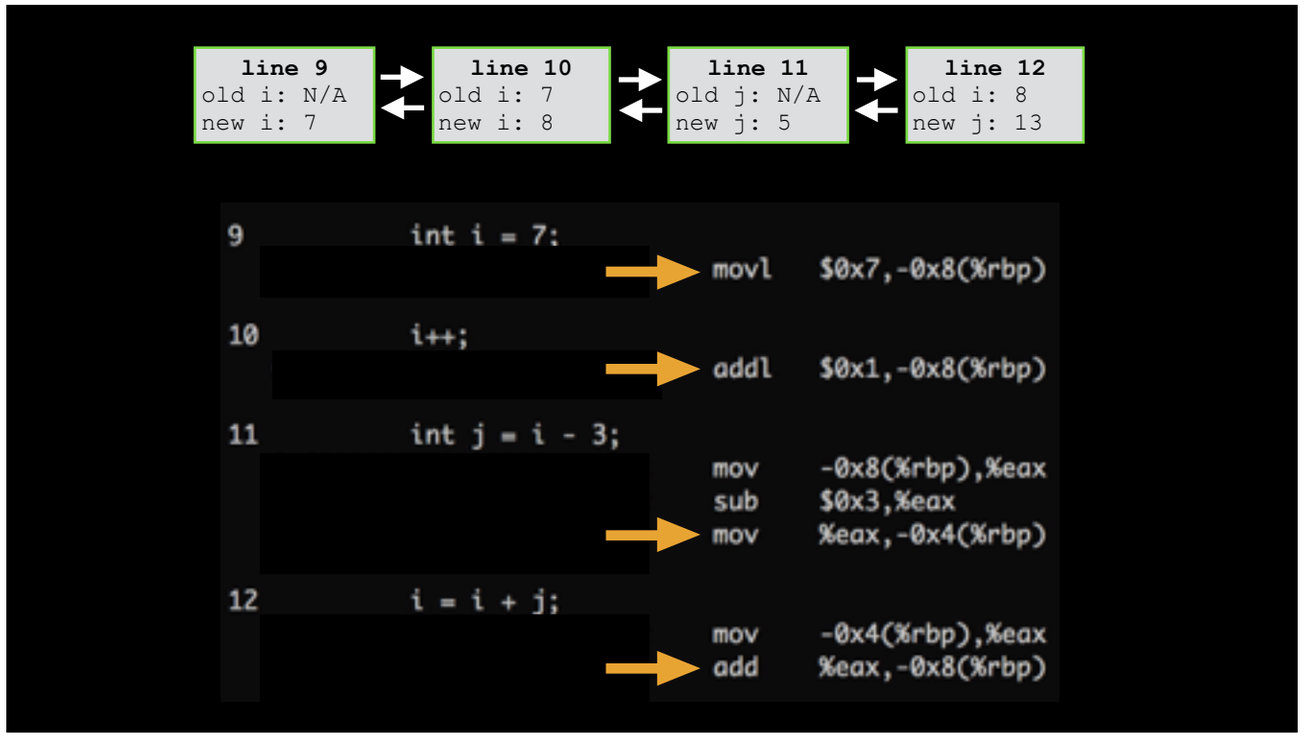
the problem is that we only have a stack trace - the functions we were currently executing when the program crashed, and the CURRENT value of that pointer. What we've lost are which functions have already returned, and previous or overwritten program state. We'd like to discover is the place where the pointer got scribbled over, but that moment has already past.
<SCROLL DOWN>
Ordinarily I would probably start setting breakpoints increasingly farther back in time in order to try and catch it, and for a small program that'll be tedious but it'll work. But for long-running programs or cases where the bug won't always be triggered, this will become really difficult. The thing that we need is a way to play execution backwards just as easily as we can play it forward. We need a

time travelling debugger to do this. At the highest level, what distinguishes an ordinary debugger from an time-traveling one is that the latter is able to reconstruct program state at any point in a process' execution. If we had an time-traveling debugger we could see where our pointer got overwritten with an erroneous value.

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

So here are a few arithmetic operations that I just made up, and <click> the corresponding x86 instructions that the compiler generated. The most straightforward way I can think of is that if we

So here are a few arithmetic operations that I just made up, and <click> the corresponding x86 instructions that the compiler generated. The most straightforward way I can think of is that if we



stored a doubly linked list of all the writes to memory that we've observed for a program run with their before and after value, then we can we can see future state of the program by reading the list forwards and past state by reading it backwards.

```
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 };
```

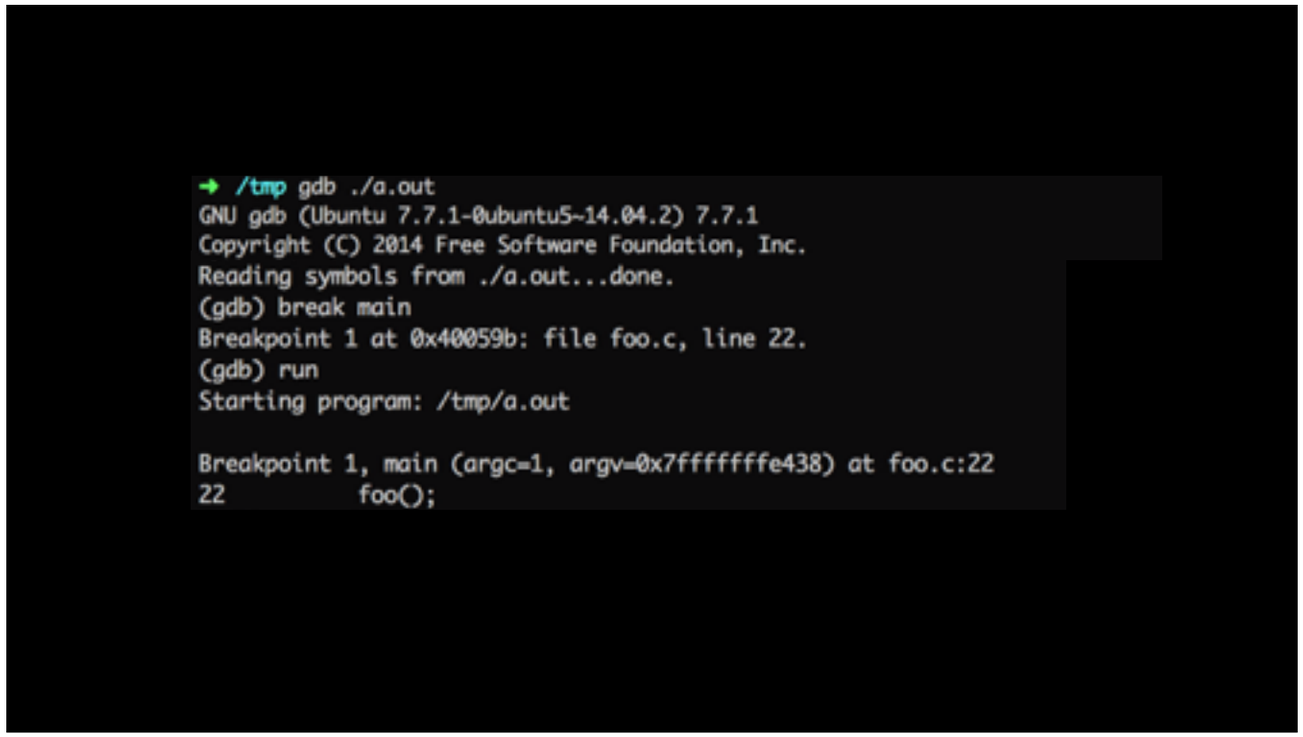
gdb/record-full.c

As it turns out, some of you may be aware that GDB, the popular open source debugger for C languages, actually supports time-traveling debugging, and this is exactly how it's implemented under the hood! When you start a time-traveling recording session, every register or memory change that the debugger observes is stored in this doubly-linked list, and stepping forward and back means single-stepping through this linked list and updating the program state. <TODO: remind why we care about GDB still>

So let’s try GDB’s reverse debugging functionality. <click> We’ll set a breakpoint at main, and then once we’ve hit the entry point to our program we’re going to <click> start recording and continue execution. So now we’ll be building that linked list in the debugger as the program executes. <click> So as expected we hit that same crash.

Ordinarily if we set a watchpoint on that variable it would be too late since the program has already crashed, but <click> if we set that watchpoint and tell time to run backwards… <click> the watch point hits the last time ptr was assigned to. And <click> in this trivial example that I cooked up, I’m assigning its value based on a NULL pointer. <click> and we can keep tracing assignments to ptr back to the start of main, where we began recording.

GDB’s time-traveling debugging is a great first example in my opinion because it goes to show that a straightforward way of building these sorts of instrumentation tooling doesn’t require much algorithmic wizardry. This means that GDB’s reverse debugging might not be as efficient as possible (indeed, my sample program informally had a slowdown of over 100x), but in the academic space,



So let's try GDB's reverse debugging functionality. <click> We'll set a breakpoint at main, and then once we've hit the entry point to our program we're going to <click> start recording and continue execution. So now we'll be building that linked list in the debugger as the program executes. <click> So as expected we hit that same crash.

Ordinarily if we set a watchpoint on that variable it would be too late since the program has already crashed, but <click> if we set that watchpoint and tell time to run backwards… <click> the watch point hits the last time ptr was assigned to. And <click> in this trivial example that I cooked up, I'm assigning its value based on a NULL pointer. <click> and we can keep tracing assignments to ptr back to the start of main, where we began recording.

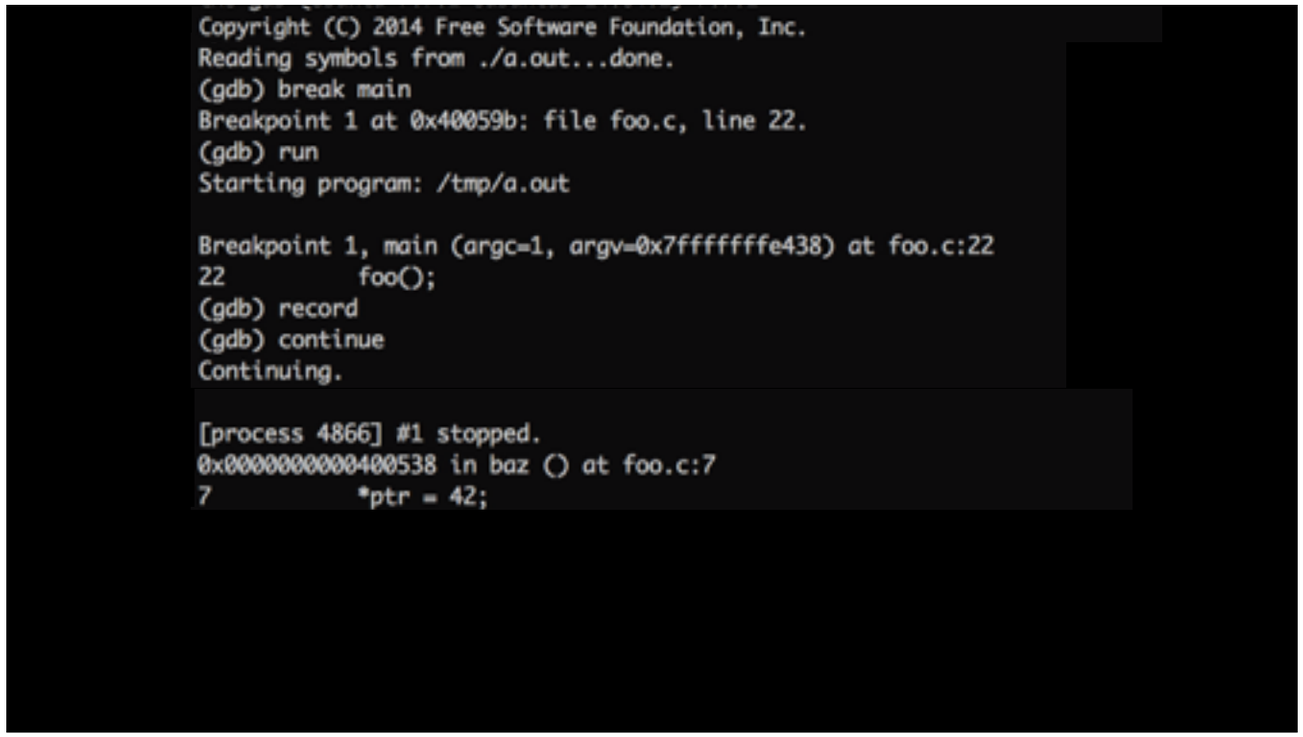
GDB's time-traveling debugging is a great first example in my opinion because it goes to show that a straightforward way of building these sorts of instrumentation tooling doesn't require much algorithmic wizardry. This means that GDB's reverse debugging might not be as efficient as possible (indeed, my sample program informally had a slowdown of over 100x), but in the academic space,

```
➜ /tmp gdb ./a.out
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
Reading symbols from ./a.out...done.
(gdb) break main
Breakpoint 1 at 0x40059b: file foo.c, line 22.
(gdb) run
Starting program: /tmp/a.out

Breakpoint 1, main (argc=1, argv=0x7fffffffe438) at foo.c:22
22          foo();
(gdb) record
(gdb) continue
Continuing.
```

So let's try GDB's reverse debugging functionality. <click> We'll set a breakpoint at main, and then once we've hit the entry point to our program we're going to <click> start recording and continue execution. So now we'll be building that linked list in the debugger as the program executes. <click> So as expected we hit that same crash.

Ordinarily if we set a watchpoint on that variable it would be too late since the program has already crashed, but <click> if we set that watchpoint and tell time to run backwards… <click> the watch point hits the last time ptr was assigned to. And <click> in this trivial example that I cooked up, I'm assigning its value based on a NULL pointer. <click> and we can keep tracing assignments to ptr back to the start of main, where we began recording.

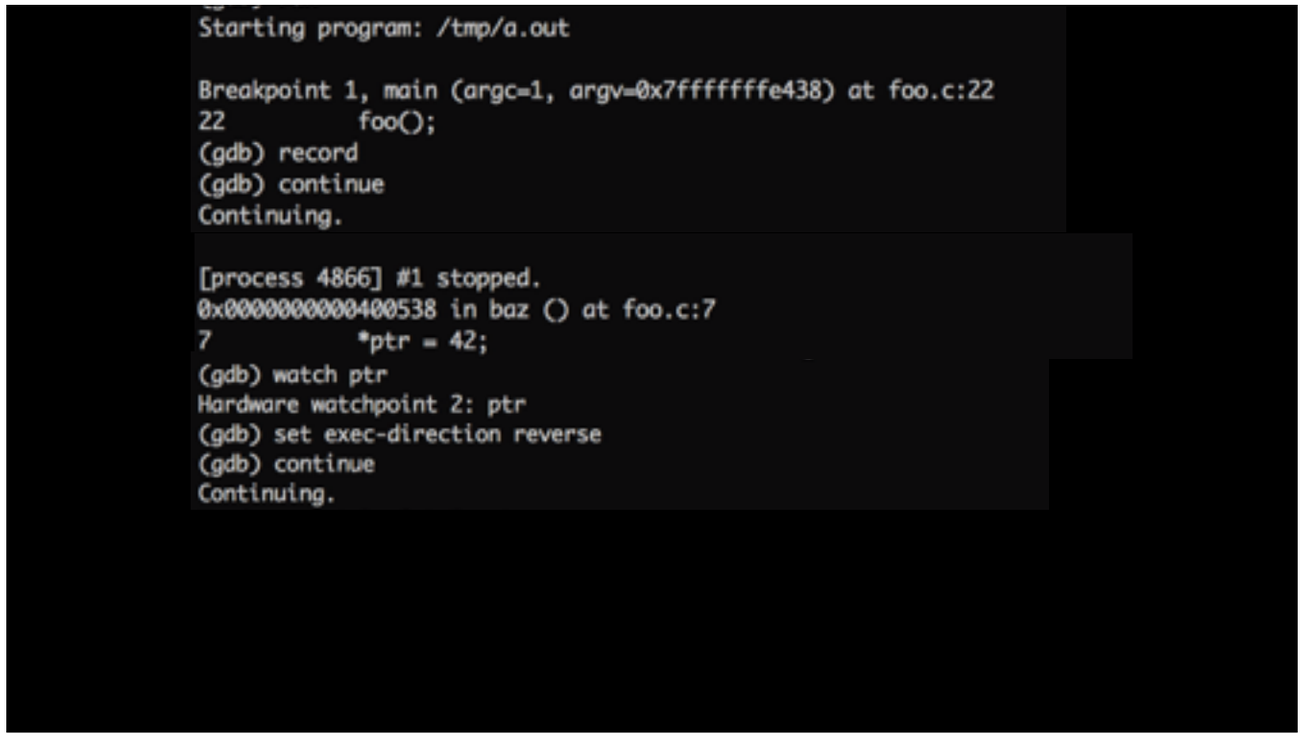
GDB's time-traveling debugging is a great first example in my opinion because it goes to show that a straightforward way of building these sorts of instrumentation tooling doesn't require much algorithmic wizardry. This means that GDB's reverse debugging might not be as efficient as possible (indeed, my sample program informally had a slowdown of over 100x), but in the academic space,



So let's try GDB's reverse debugging functionality. <click> We'll set a breakpoint at main, and then once we've hit the entry point to our program we're going to <click> start recording and continue execution. So now we'll be building that linked list in the debugger as the program executes. <click> So as expected we hit that same crash.

Ordinarily if we set a watchpoint on that variable it would be too late since the program has already crashed, but <click> if we set that watchpoint and tell time to run backwards… <click> the watch point hits the last time ptr was assigned to. And <click> in this trivial example that I cooked up, I'm assigning its value based on a NULL pointer. <click> and we can keep tracing assignments to ptr back to the start of main, where we began recording.

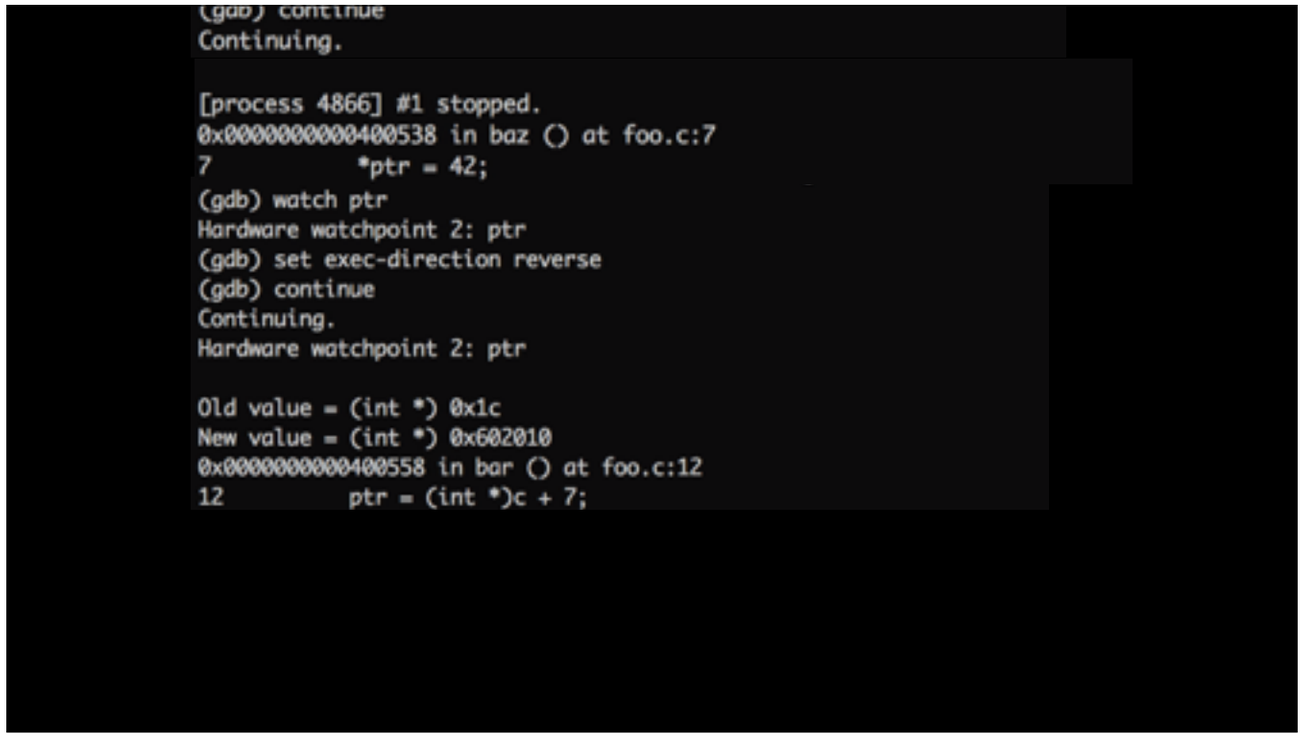
GDB's time-traveling debugging is a great first example in my opinion because it goes to show that a straightforward way of building these sorts of instrumentation tooling doesn't require much algorithmic wizardry. This means that GDB's reverse debugging might not be as efficient as possible (indeed, my sample program informally had a slowdown of over 100x), but in the academic space,



So let's try GDB's reverse debugging functionality. <click> We'll set a breakpoint at main, and then once we've hit the entry point to our program we're going to <click> start recording and continue execution. So now we'll be building that linked list in the debugger as the program executes. <click> So as expected we hit that same crash.

Ordinarily if we set a watchpoint on that variable it would be too late since the program has already crashed, but <click> if we set that watchpoint and tell time to run backwards… <click> the watch point hits the last time ptr was assigned to. And <click> in this trivial example that I cooked up, I'm assigning its value based on a NULL pointer. <click> and we can keep tracing assignments to ptr back to the start of main, where we began recording.

GDB's time-traveling debugging is a great first example in my opinion because it goes to show that a straightforward way of building these sorts of instrumentation tooling doesn't require much algorithmic wizardry. This means that GDB's reverse debugging might not be as efficient as possible (indeed, my sample program informally had a slowdown of over 100x), but in the academic space,



So let's try GDB's reverse debugging functionality. <click> We'll set a breakpoint at main, and then once we've hit the entry point to our program we're going to <click> start recording and continue execution. So now we'll be building that linked list in the debugger as the program executes. <click> So as expected we hit that same crash.

Ordinarily if we set a watchpoint on that variable it would be too late since the program has already crashed, but <click> if we set that watchpoint and tell time to run backwards… <click> the watch point hits the last time ptr was assigned to. And <click> in this trivial example that I cooked up, I'm assigning its value based on a NULL pointer. <click> and we can keep tracing assignments to ptr back to the start of main, where we began recording.

GDB's time-traveling debugging is a great first example in my opinion because it goes to show that a straightforward way of building these sorts of instrumentation tooling doesn't require much algorithmic wizardry. This means that GDB's reverse debugging might not be as efficient as possible (indeed, my sample program informally had a slowdown of over 100x), but in the academic space,

```
(gdb) watch ptr
Hardware watchpoint 2: ptr
(gdb) set exec-direction reverse
(gdb) continue
Continuing.
Hardware watchpoint 2: ptr

Old value = (int *) 0x1c
New value = (int *) 0x602010
0x0000000000400558 in bar () at foo.c:12
12          ptr = (int *)c + 7;
(gdb) inf loc
c = 0x0
(gdb) continue
Continuing.
Hardware watchpoint 2: ptr
```

So let's try GDB's reverse debugging functionality. <click> We'll set a breakpoint at main, and then once we've hit the entry point to our program we're going to <click> start recording and continue execution. So now we'll be building that linked list in the debugger as the program executes. <click> So as expected we hit that same crash.

Ordinarily if we set a watchpoint on that variable it would be too late since the program has already crashed, but <click> if we set that watchpoint and tell time to run backwards… <click> the watch point hits the last time ptr was assigned to. And <click> in this trivial example that I cooked up, I'm assigning its value based on a NULL pointer. <click> and we can keep tracing assignments to ptr back to the start of main, where we began recording.

GDB's time-traveling debugging is a great first example in my opinion because it goes to show that a straightforward way of building these sorts of instrumentation tooling doesn't require much algorithmic wizardry. This means that GDB's reverse debugging might not be as efficient as possible (indeed, my sample program informally had a slowdown of over 100x), but in the academic space,

```
(gdb) inf loc
c = 0x0
(gdb) continue
Continuing.
Hardware watchpoint 2: ptr

Old value = (int *) 0x602010
New value = (int *) 0x0
0x0000000000400579 in foo () at foo.c:17
17          ptr = malloc(256);
(gdb)
Continuing.

No more reverse-execution history.
main (argc=1, argv=0x7fffffffe438) at foo.c:22
22          foo();
(gdb)
```

So let's try GDB's reverse debugging functionality. <click> We'll set a breakpoint at main, and then once we've hit the entry point to our program we're going to <click> start recording and continue execution. So now we'll be building that linked list in the debugger as the program executes. <click> So as expected we hit that same crash.

Ordinarily if we set a watchpoint on that variable it would be too late since the program has already crashed, but <click> if we set that watchpoint and tell time to run backwards… <click> the watch point hits the last time ptr was assigned to. And <click> in this trivial example that I cooked up, I'm assigning its value based on a NULL pointer. <click> and we can keep tracing assignments to ptr back to the start of main, where we began recording.

GDB's time-traveling debugging is a great first example in my opinion because it goes to show that a straightforward way of building these sorts of instrumentation tooling doesn't require much algorithmic wizardry. This means that GDB's reverse debugging might not be as efficient as possible (indeed, my sample program informally had a slowdown of over 100x), but in the academic space,

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 UNDUCK 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 [DD99Ø2], 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 [LH99Ø].

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

a lot of effort is spent researching what the user interface and visual presentation should be for these tools, rather than how to necessarily make them super fast. So here’s a paper <click> that spends more than a few pages on how best to present the information to a user. There are lots of papers that do this. Given how mind-bending reverse debugging can be, I don’t think this is necessarily a bad thing but it means there’s still opportunity for improvement.

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

NODe ‘06

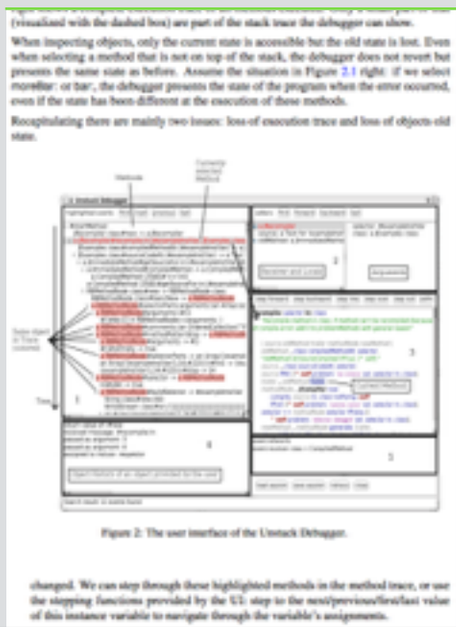
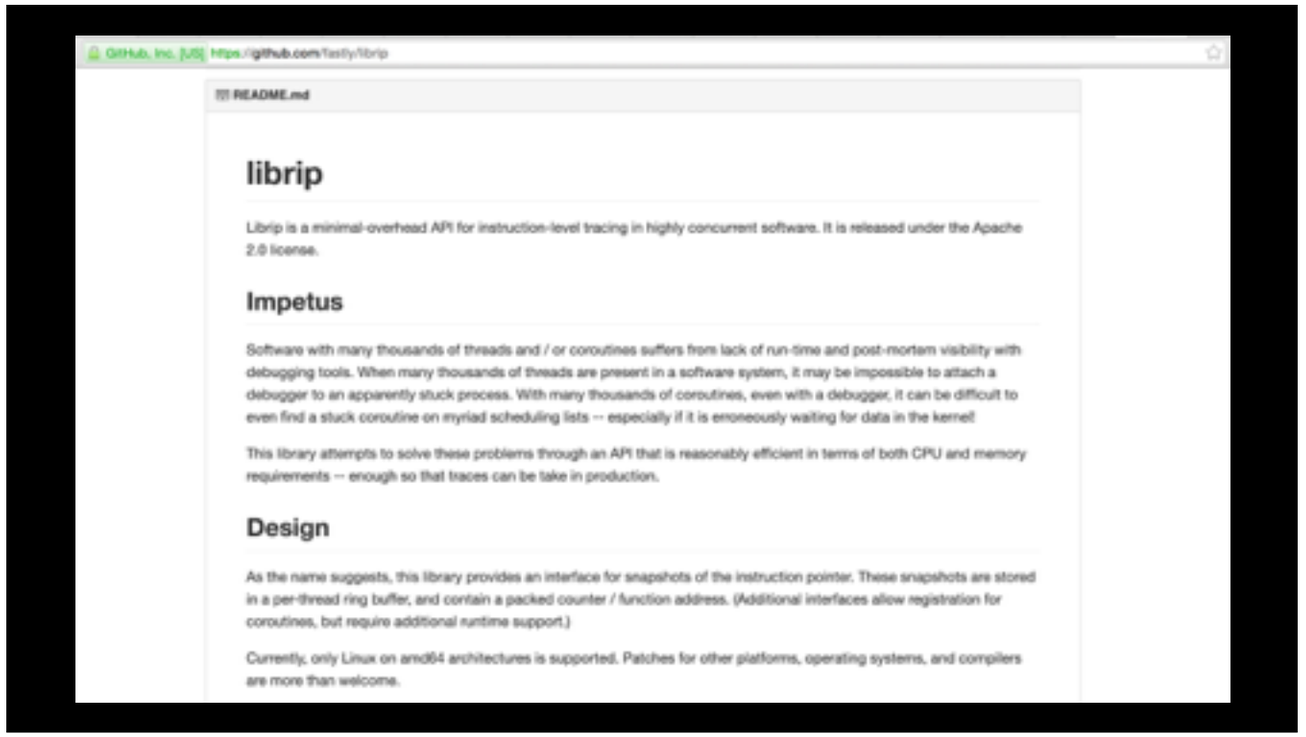


Figure 2: The user interface of the Unstack Debugger.

changed. We can step through these highlighted methods in the method trace, or use the stepping functions provided by the UI: step to the next/previous/first/last value of this instance variable to navigate through the variable's assignments.

Figure 3: Left: highlight the modifiers of a variable over the context menu in the source view. Right: the result of the action made on the left side: highlighted methods in which var2 was modified (i.e., changed the value)

a lot of effort is spent researching what the user interface and visual presentation should be for these tools, rather than how to necessarily make them super fast. So here’s a paper <click> that spends more than a few pages on how best to present the information to a user. There are lots of papers that do this. Given how mind-bending reverse debugging can be, I don’t think this is necessarily a bad thing but it means there’s still opportunity for improvement.



I do want to point out that Fastly, today, uses a simplified form of this debugging technique to answer the “where were we before we crashed” question. We don’t track memory writes but we do track relevant control flow changes using a small library we open sourced called librip. Even though it does way less than a proper time-traveling debugger would it’s been invaluable for us to understand what the system was doing right before a crash, which is a big help with bugsquashing.

But, that said, let’s talk a bit about how better we could store these execution data structures. GDB’s simplistic approach might make sense if you’re not recording for very long, and are mostly just stepping between a few instructions, but for long-running recordings we might want to do something different. This is where a bunch of work from the academic literature comes into play.

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[*] 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 in 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 repeats.

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-

One of my favourite papers is Sam King’s Time-Traveling Virtual Machines paper, where he talks about efficiently building time-traveling debuggers for entire operating systems. A relevant observation is made right at the start that <click> the class of bugs the authors are interested in require potentially weeks or months of execution before the bug is triggered, so the GDB style linked list for every memory operation over weeks of execution isn’t feasible.

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[*] 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 repeats.

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-

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

One of my favourite papers is Sam King’s Time-Traveling Virtual Machines paper, where he talks about efficiently building time-traveling debuggers for entire operating systems. A relevant observation is made right at the start that <click> the class of bugs the authors are interested in require potentially weeks or months of execution before the bug is triggered, so the GDB style linked list for every memory operation over weeks of execution isn’t feasible.

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[*] 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 repeats.

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.

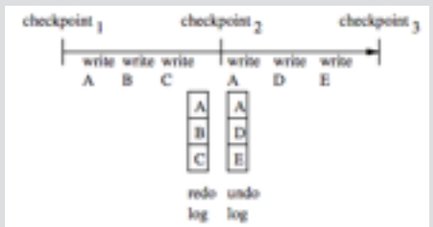
Third, the act of debugging may perturb the state of the operating system. The converse is also true: a misbehaving operating system may corrupt the state of the debugger. These interactions are possible because the operating system is traditionally the lowest level of software on a computer, so the 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-

So the authors present two ideas that in my opinion are pretty fundamental. <click> The first is that the authors avoid having to replay every write operation from scratch by periodically checkpointing the state of the system, the frequency of which depends on whether a user is attached to the debugger working interactively or whether it’s just recording execution for future analysis. Each checkpoint contains essentially a diff of the state before or after it - the “redo” log contains what changed from the previous checkpoint and the “undo” log contains what will change between now and the next checkpoint. Notice that this seems similar to what GDB’s linked list has, but it’s over an epoch spanning multiple seconds rather than every single memory write. But don’t we still need to store every write if we want to step back and forth between timestamps?

King et al. **Debugging Operating Systems with Time-Traveling Virtual Machines** Usenix ‘05



So the authors present two ideas that in my opinion are pretty fundamental. <click> The first is that the authors avoid having to replay every write operation from scratch by periodically checkpointing the state of the system, the frequency of which depends on whether a user is attached to the debugger working interactively or whether it’s just recording execution for future analysis. Each checkpoint contains essentially a diff of the state before or after it - the “redo” log contains what changed from the previous checkpoint and the “undo” log contains what will change between now and the next checkpoint. Notice that this seems similar to what GDB’s linked list has, but it’s over an epoch spanning multiple seconds rather than every single memory write. But don’t we still need to store every write if we want to step back and forth between timestamps?

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 TTVMs 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 [8]. 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 [3, 8]. 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 [18]. 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—if 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

Well the authors observe that <click> replaying an execution means, by definition, that the machine is going to do the same thing that it did when you were recording. It’s deterministic. It turns out <click> that the examples of operations that aren’t functions of the current state of the system are external events like network traffic, user input, and hardware interrupts, so those are the things you have to record.

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 TTVMs 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. Intel 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 [8]. 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 [3, 8]. 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 reprovisioning 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 reprovisioned.

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 [18]. 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”

Well the authors observe that <click> replaying an execution means, by definition, that the machine is going to do the same thing that it did when you were recording. It’s deterministic. It turns out <click> that the examples of operations that aren’t functions of the current state of the system are external events like network traffic, user input, and hardware interrupts, so those are the things you have to record.

King et al. **Debugging Operating Systems with Time-Traveling Virtual Machines** Usenix '05

"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"

Well the authors observe that <click> replaying an execution means, by definition, that the machine is going to do the same thing that it did when you were recording. It's deterministic. It turns out <click> that the examples of operations that aren't functions of the current state of the system are external events like network traffic, user input, and hardware interrupts, so those are the things you have to record.

Deterministic Replay

This super important property is referred to in the literature as “deterministic replay” and I would argue it’s one of the things that makes dynamic analysis practical for production use, particularly when we start looking at multiprocessor systems. But, for this work, how practical is practical?

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

Usenix ‘05

Unsurprisingly, it depends on how often you checkpoint. Logging along adds single-digit performance overhead and negligible storage requirements, but taking checkpoints frequently enough to make interactive debugging possible, in the worst case, slowed things down by a third. Good enough for you to run production traffic on? You be the judge.

King et al. **Debugging Operating Systems with Time-Traveling Virtual Machines** Usenix '05

"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"

Unsurprisingly, it depends on how often you checkpoint. Logging along adds single-digit performance overhead and negligible storage requirements, but taking checkpoints frequently enough to make interactive debugging possible, in the worst case, slowed things down by a third. Good enough for you to run production traffic on? You be the judge.

- 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

So we talked about how GDB was slow because it stored every memory operation. That's true but that's only one part of the story - it's also slow because it has to single-step through the program in order to replay the memory write log. That's one of the reasons why GDB was so tremendously slow; we could only execute one instruction before control flow would have to be returned to the debugger.

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

So rather than storing a record of how the program mutates itself over time, let’s have the program recompile itself with instrumention weaved into the code segment. Not only does this create a more performant analysis tool, but it often enables more higher-level semantics around the kinds of tools or analyses you might want to perform. Today we’re going to contrast two such 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, E-mail: pin.project@intel.com

Abstract

Robust and powerful software instrumentation tools are essential for program analysis tools 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. Pin 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. To the best of our knowledge, 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

1. Introduction

The first tool is called Pin, which came out of Intel and the University of Colorado, and had a PLDI paper published about it. It’s also available for download from Intel’s site. Right from the start <click> the authors talk about the generality of the tool - it’s not a debugger, but rather a platform for building 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 V.Jay Janapa Reddi † Kim Hazelwood

Intel Corporation †University of Colorado

Abstract

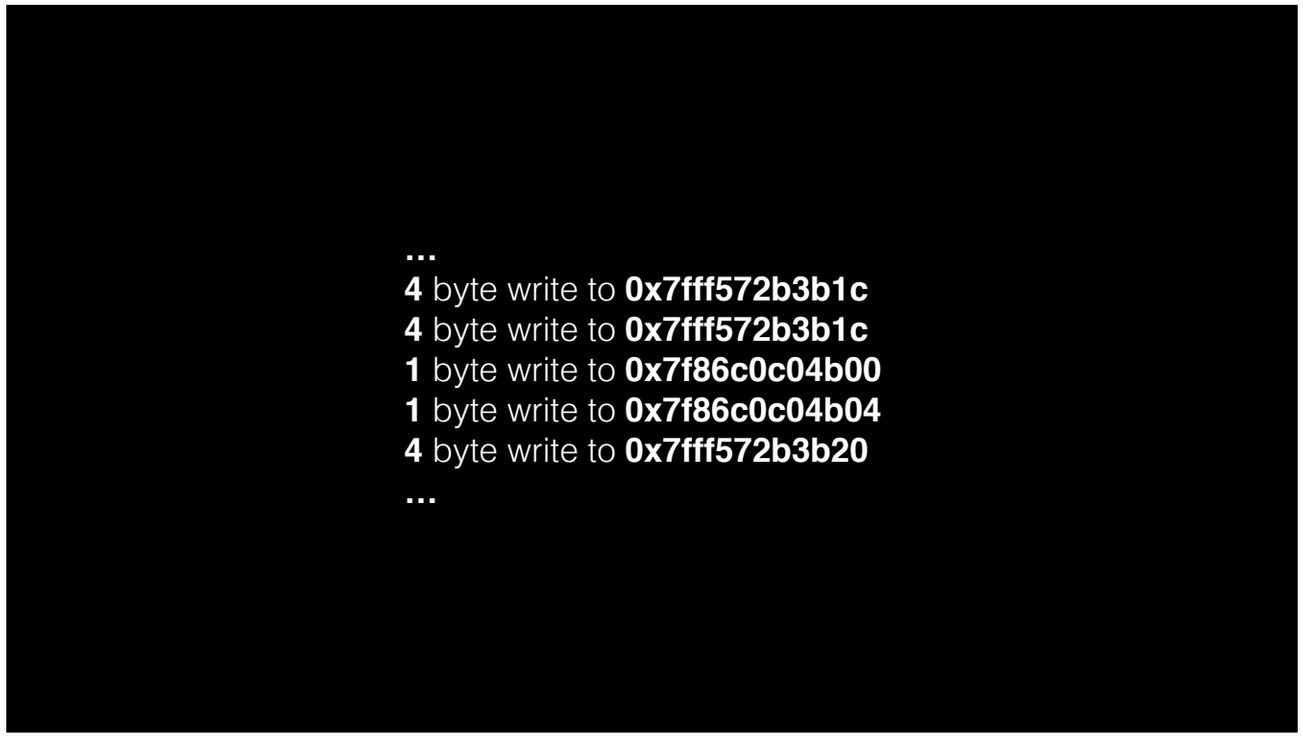
1. Introduction

"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"

The first tool is called Pin, which came out of Intel and the University of Colorado, and had a PLDI paper published about it. It's also available for download from Intel's site. Right from the start <click> the authors talk about the generality of the tool - it's not a debugger, but rather a platform for building analysis tools.



So imagine that we wanted to write a dynamic analysis tool that, just for instructional purposes, simply prints a line every time the instrumented program writes to memory.

Luk et al. **Pin: Building Customized Program Analysis Tools w/ Dynamic Instrumentation** PLDI ‘05


Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation

So with Pin, the user builds an analysis tool in C - this is a simple tool that invokes a callback for every memory write, so the Instrument function is invoked at compile time and inserts a runtime call to RecordMemWrite. So, in <click> our example of memory operations from before, <click> we have four memory writes, so we’ll insert four calls to our instrument function, where we’ll pass it the current instruction pointer value and the address of the write.

Luk et al. **Pin: Building Customized Program Analysis Tools w/ Dynamic Instrumentation** PLDI ‘05

```
FILE * trace;

// Print a memory write record
VOID RecordMemWrite(VOID * ip, VOID * addr, UINT32 size)
    fprintf(trace,"%p: W %p %d\n", ip, addr, size);
}

// Called for every instruction
VOID Instruction(INS ins, VOID *v) {
    // instruments writes using a predicated call,
    // i.e. the call happens iff the store is
    // actually executed
    if (INS_IsMemoryWrite(ins))
        INS_InsertPredicatedCall(
            ins, IPOINT_BEFORE, AFUNPTR(RecordMemWrite),
            IARG_INST_PTR, IARG_MEMORYWRITE_EA,
            IARG_MEMORYWRITE_SIZE, IARG_END);
}

int main(int argc, char *argv[]) {
    PIN_Init(argc, argv);
    trace = fopen("strace.out", "w");
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_StartProgram(); // Never returns
    return 0;
}
```

Figure 1. A Pintool for tracing memory writes.

So with Pin, the user builds an analysis tool in C - this is a simple tool that invokes a callback for every memory write, so the Instrument function is invoked at compile time and inserts a runtime call to RecordMemWrite. So, in <click> our example of memory operations from before, <click> we have four memory writes, so we’ll insert four calls to our instrument function, where we’ll pass it the current instruction pointer value and the address of the write.

Luk et al. **Pin: Building Customized Program Analysis Tools w/ Dynamic Instrumentation** PLDI ‘05

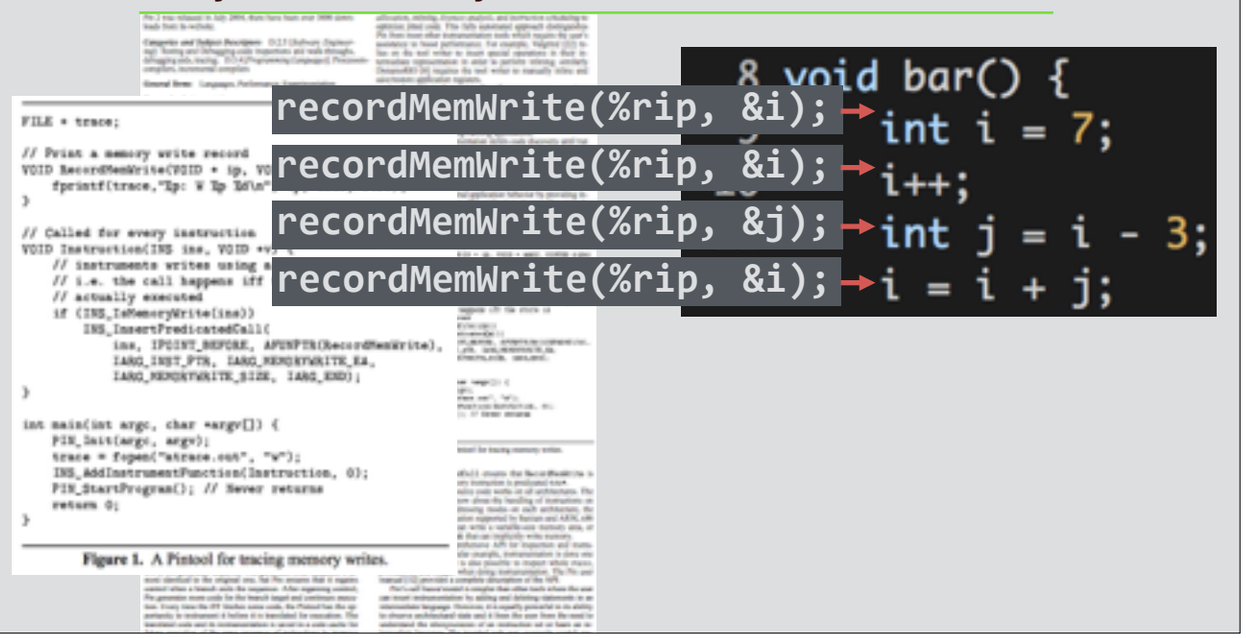
```
FILE * trace;

// Print a memory write record
VOID RecordMemWrite(VOID * ip, VOID * addr, UINT32 size)
    fprintf(trace,"%p: W %p %d\n", ip, addr, size);
}

// Called for every instruction
VOID Instruction(INS ins, VOID *v) {
    // instruments writes using a predicated call,
    // i.e. the call happens iff the store is
    // actually executed
    if (INS_IsMemoryWrite(ins))
        INS_InsertPredicatedCall(
            ins, IPOINT_BEFORE, AFUNPTR(RecordMemWrite),
            IARG_INST_PTR, IARG_MEMORYWRITE_EA,
            IARG_MEMORYWRITE_SIZE, IARG_END);
}

int main(int argc, char *argv[]) {
    PIN_Init(argc, argv);
    trace = fopen("atrace.out", "w");
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_StartProgram(); // Never returns
    return 0;
}
```

Figure 1. A Pintool for tracing memory writes.

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

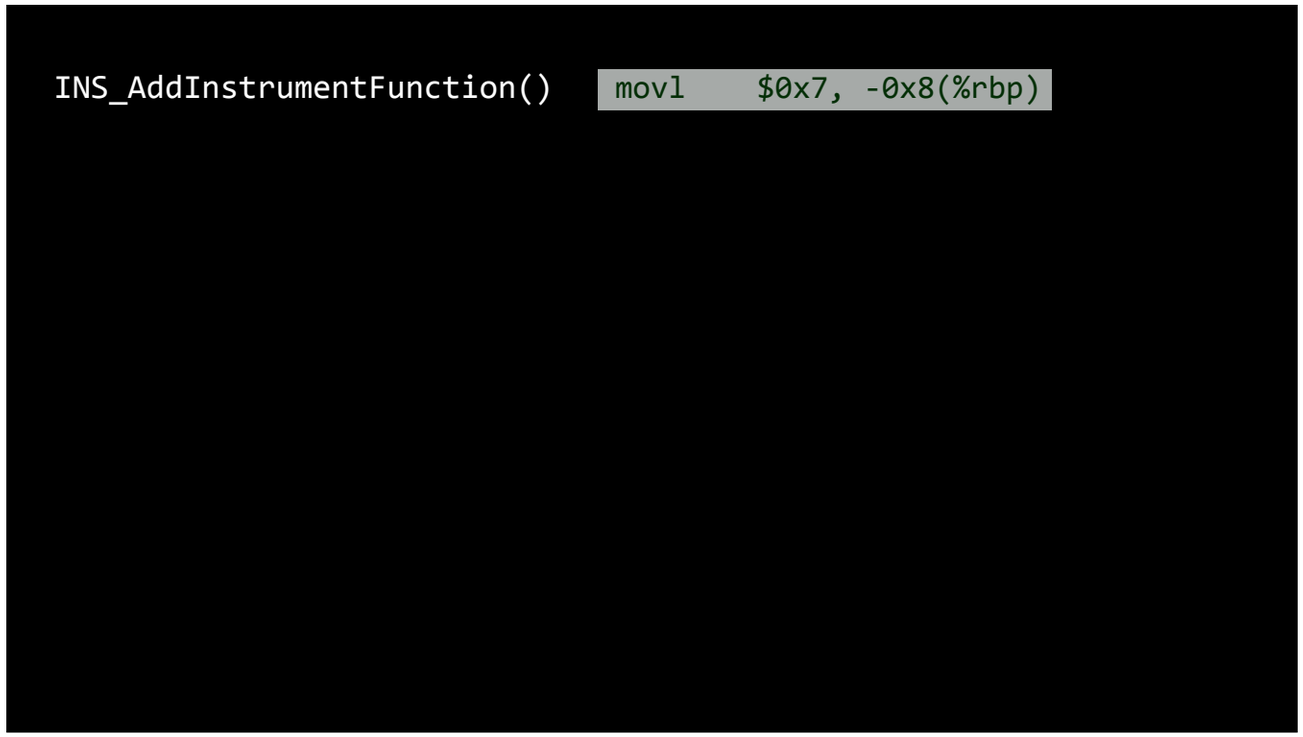
So with Pin, the user builds an analysis tool in C - this is a simple tool that invokes a callback for every memory write, so the Instrument function is invoked at compile time and inserts a runtime call to RecordMemWrite. So, in <click> our example of memory operations from before, <click> we have four memory writes, so we’ll insert four calls to our instrument function, where we’ll pass it the current instruction pointer value and the address of the write.

Luk et al. **Pin: Building Customized Program Analysis Tools w/ Dynamic Instrumentation** PLDI ‘05



So with Pin, the user builds an analysis tool in C - this is a simple tool that invokes a callback for every memory write, so the Instrument function is invoked at compile time and inserts a runtime call to RecordMemWrite. So, in <click> our example of memory operations from before, <click> we have four memory writes, so we’ll insert four calls to our instrument function, where we’ll pass it the current instruction pointer value and the address of the write.

So Pin has four basic primitives that you can specify instrumentation for: <click> individual instructions, like we saw with memory writes, <click> what Pin calls “traces”, which are basic blocks - sequences of instructions that don’t have any control flow changes, <click> whole functions or routines, or <click> “images”, which is the whole program. The last one is useful if you want to instrument, say, all calls to a function from all points in the program. All in all this is a pretty low-level API.



So Pin has four basic primitives that you can specify instrumentation for: <click> individual instructions, like we saw with memory writes, <click> what Pin calls “traces”, which are basic blocks - sequences of instructions that don’t have any control flow changes, <click> whole functions or routines, or <click> “images”, which is the whole program. The last one is useful if you want to instrument, say, all calls to a function from all points in the program. All in all this is a pretty low-level API.

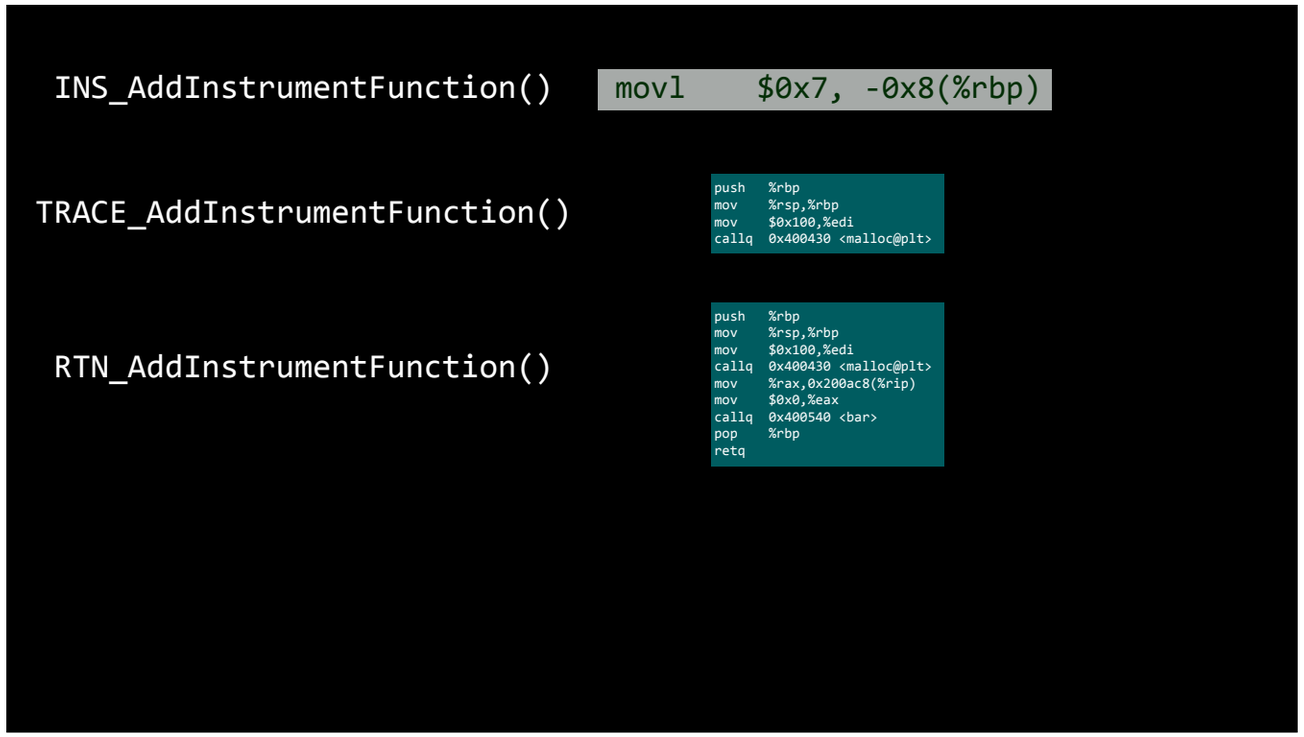
INS_AddInstrumentFunction()

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

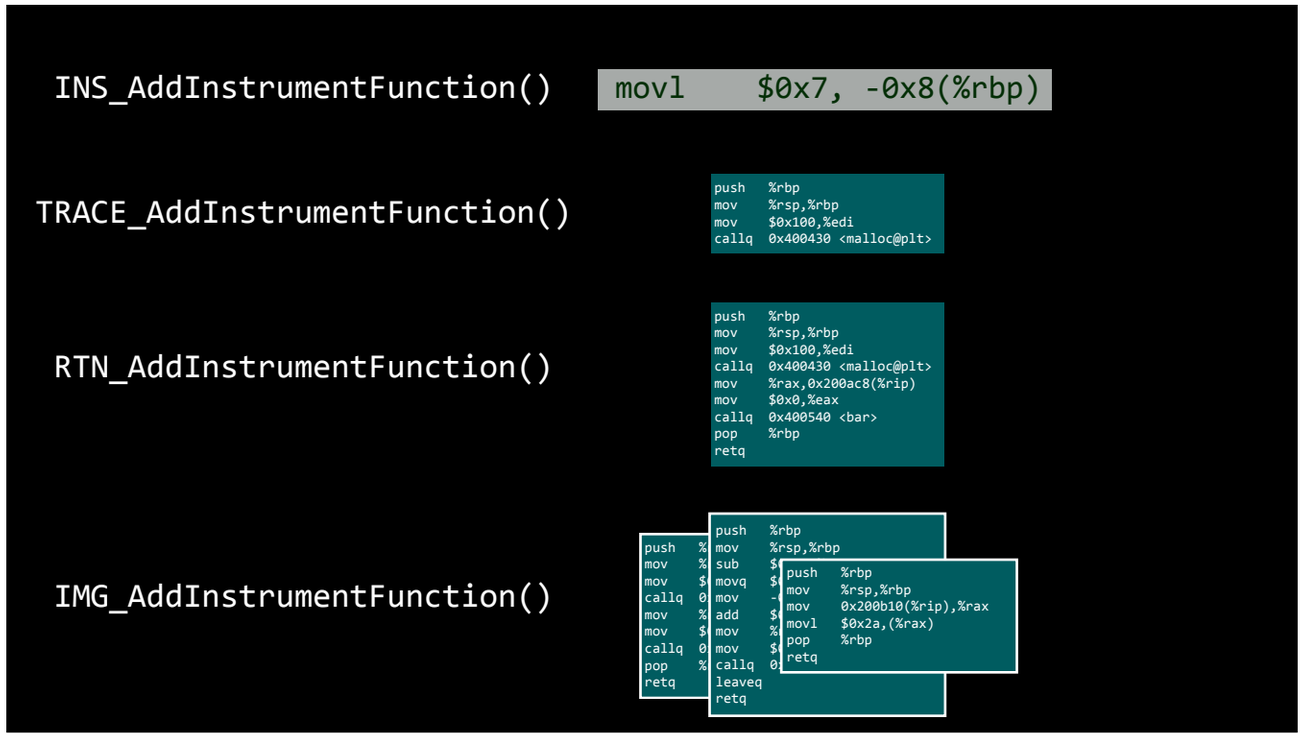
TRACE_AddInstrumentFunction()

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

So Pin has four basic primitives that you can specify instrumentation for: <click> individual instructions, like we saw with memory writes, <click> what Pin calls “traces”, which are basic blocks - sequences of instructions that don’t have any control flow changes, <click> whole functions or routines, or <click> “images”, which is the whole program. The last one is useful if you want to instrument, say, all calls to a function from all points in the program. All in all this is a pretty low-level API.



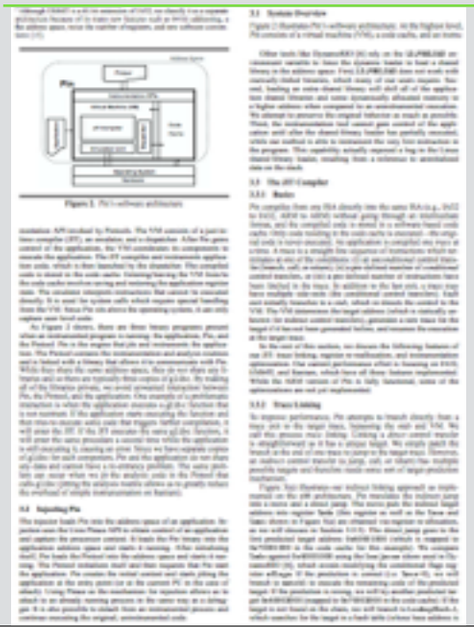
So Pin has four basic primitives that you can specify instrumentation for: <click> individual instructions, like we saw with memory writes, <click> what Pin calls “traces”, which are basic blocks - sequences of instructions that don’t have any control flow changes, <click> whole functions or routines, or <click> “images”, which is the whole program. The last one is useful if you want to instrument, say, all calls to a function from all points in the program. All in all this is a pretty low-level API.



So Pin has four basic primitives that you can specify instrumentation for: <click> individual instructions, like we saw with memory writes, <click> what Pin calls “traces”, which are basic blocks - sequences of instructions that don’t have any control flow changes, <click> whole functions or routines, or <click> “images”, which is the whole program. The last one is useful if you want to instrument, say, all calls to a function from all points in the program. All in all this is a pretty low-level API.

This API seems very reasonable but I think it’s worth pointing out how much effort you need in order to build a similar thing in a conventional compiler - this is the code you’d need to write a similar thing inside GCC. You can see how much more effort it requires, because here we have a general-purpose compiler API and are tightly coupled with GCC’s abstract syntax tree, whereas with Pin you only care about things at the level of each CPU instruction. Domain-specific tools like Pin abstract a lot of this boilerplate away, which is invaluable if you’re building a complicated tool.

Luk et al. **Pin: Building Customized Program Analysis Tools w/ Dynamic Instrumentation** PLDI ‘05



So, going back to Pinn, Pin’s JIT compiler translates native code directly back to native code by copying the code segment into a new memory region and adding programmer-defined instrumentation.

Luk et al. **Pin: Building Customized Program Analysis Tools w/ Dynamic Instrumentation** PLDI ‘05

“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.”

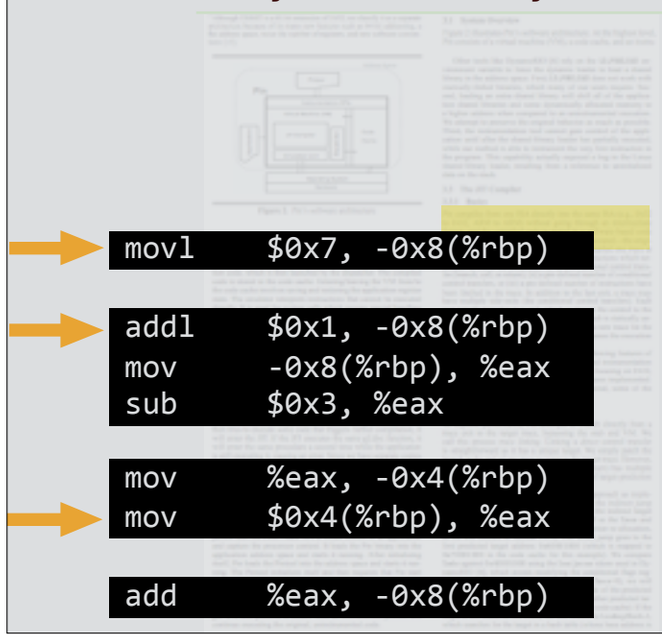
So, going back to Pinn, Pin’s JIT compiler translates native code directly back to native code by copying the code segment into a new memory region and adding programmer-defined instrumentation.

Luk et al. **Pin: Building Customized Program Analysis Tools w/ Dynamic Instrumentation** PLDI '05

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

So, starting with an instruction stream on the left, we might end up with something on the right in a newly-generated code segment.

Luk et al. **Pin: Building Customized Program Analysis Tools w/ Dynamic Instrumentation** PLDI '05

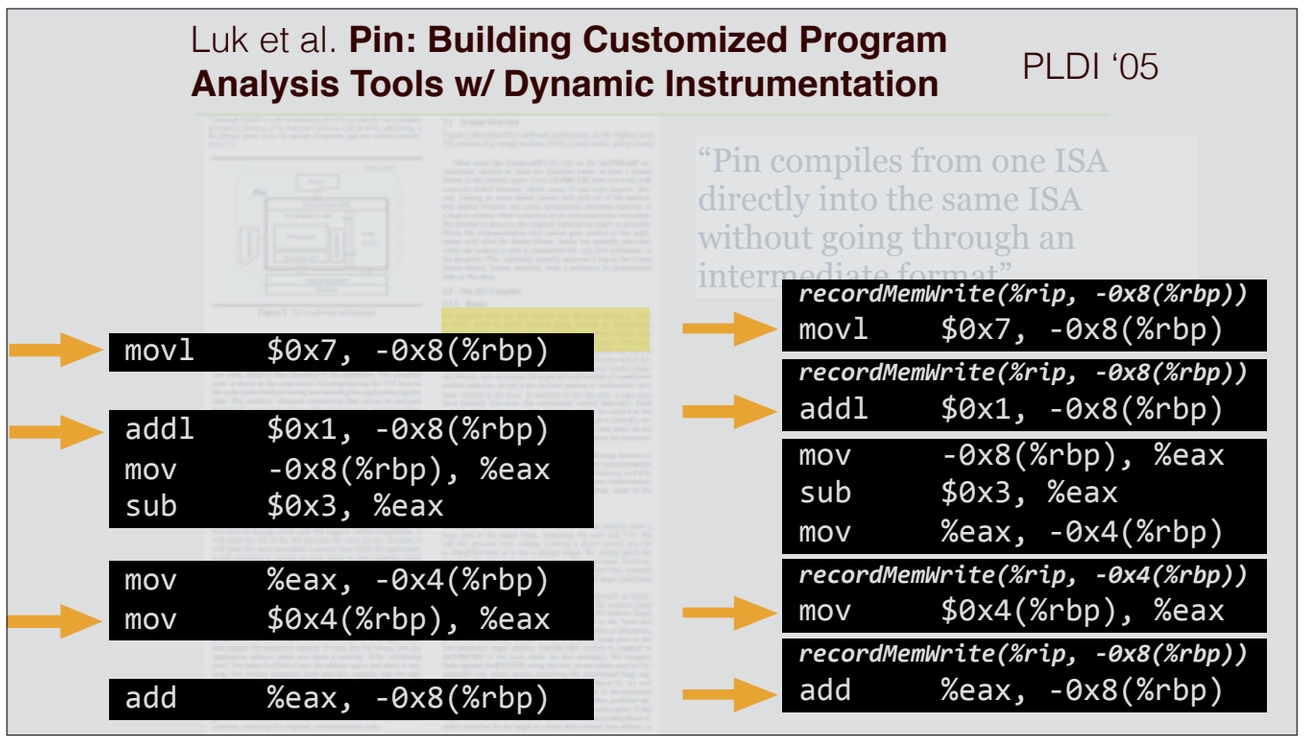


"Pin compiles from one ISA directly into the same ISA without going through an intermediate format"

```
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)
```

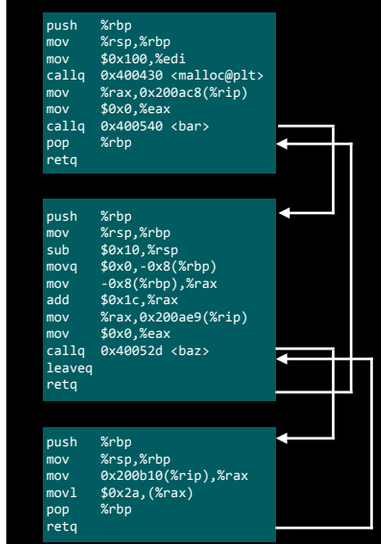
So, starting with an instruction stream on the left, we might end up with something on the right in a newly-generated code segment.

Luk et al. **Pin: Building Customized Program Analysis Tools w/ Dynamic Instrumentation** PLDI '05

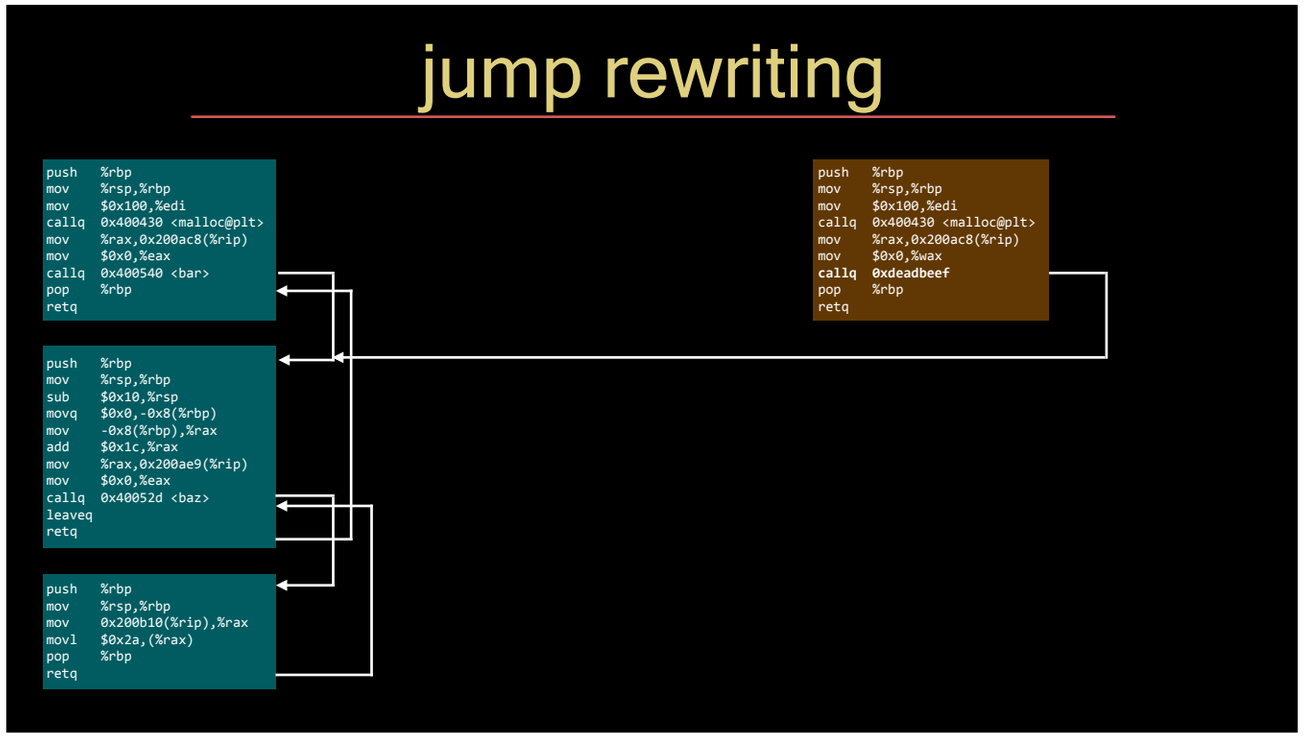


So, starting with an instruction stream on the left, we might end up with something on the right in a newly-generated code segment.

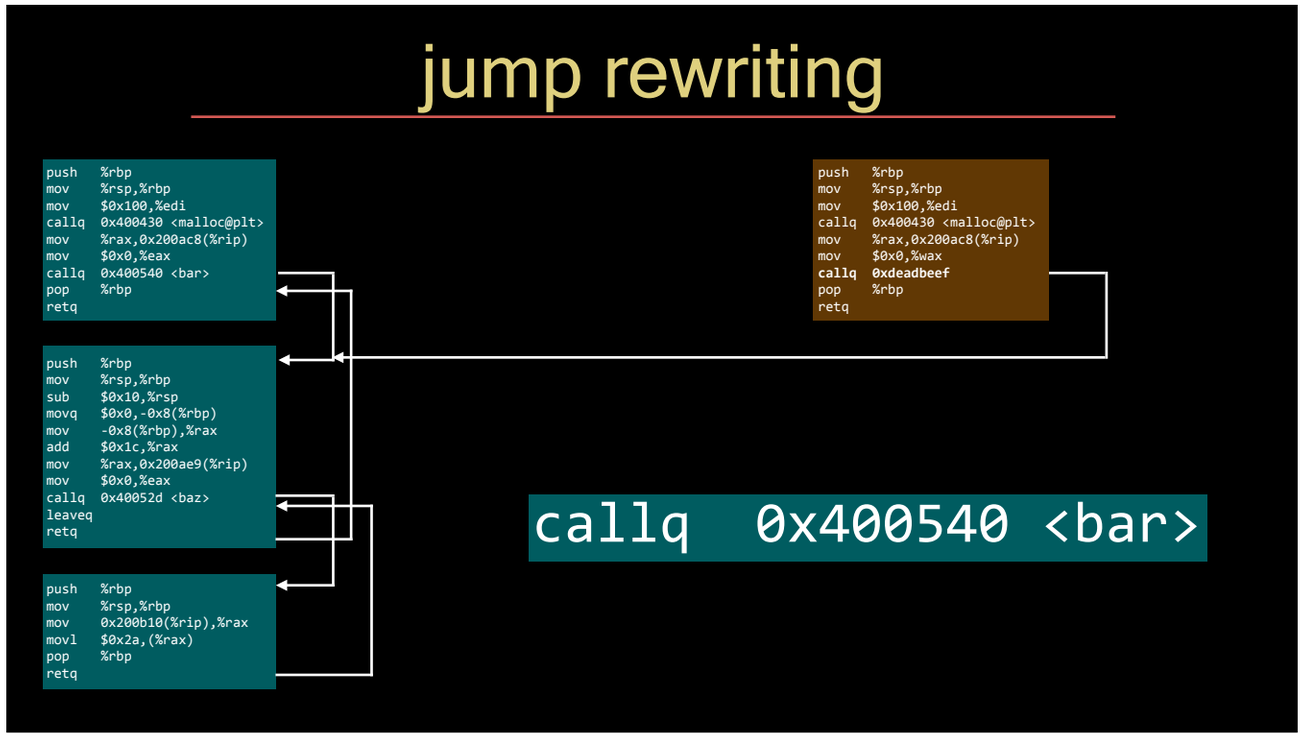
code rewriting



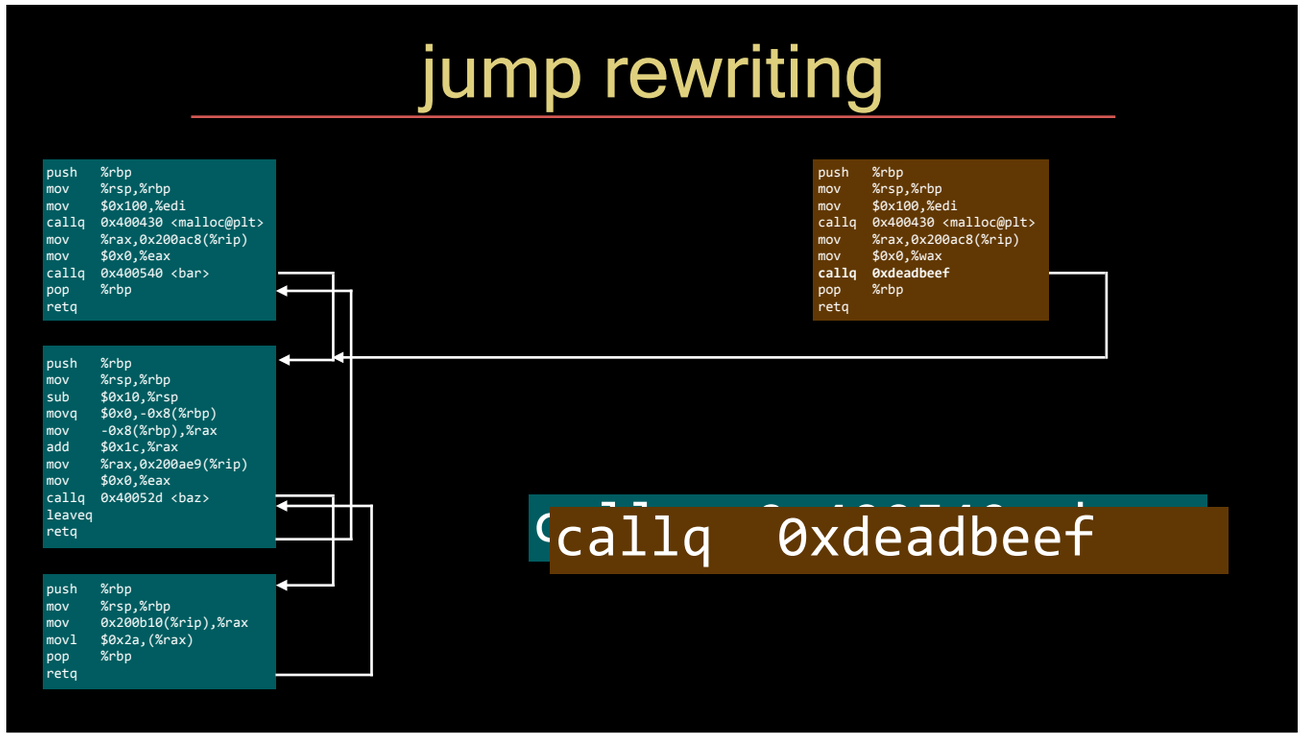
This rewriting happens lazily when we first begin to execute a block of code. So let’s say it’s the first time we’re entering the top block -



We copy the block and instrument it, but if we don’t patch that control flow change, we could find ourselves jumping back into uninstrumented code! So what we do we <click> take the target address and <click> have it call back into Pin, which would in turn

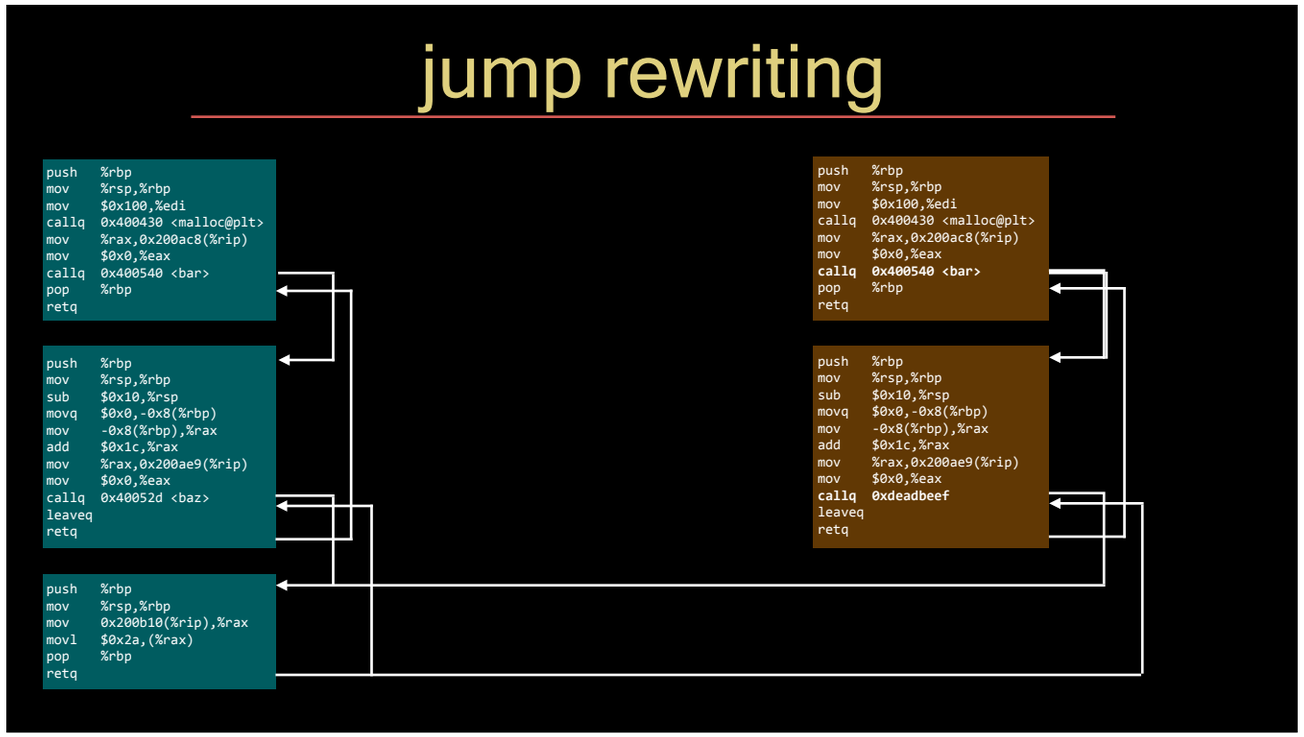


We copy the block and instrument it, but if we don’t patch that control flow change, we could find ourselves jumping back into uninstrumented code! So what we do we <click> take the target address and <click> have it call back into Pin, which would in turn



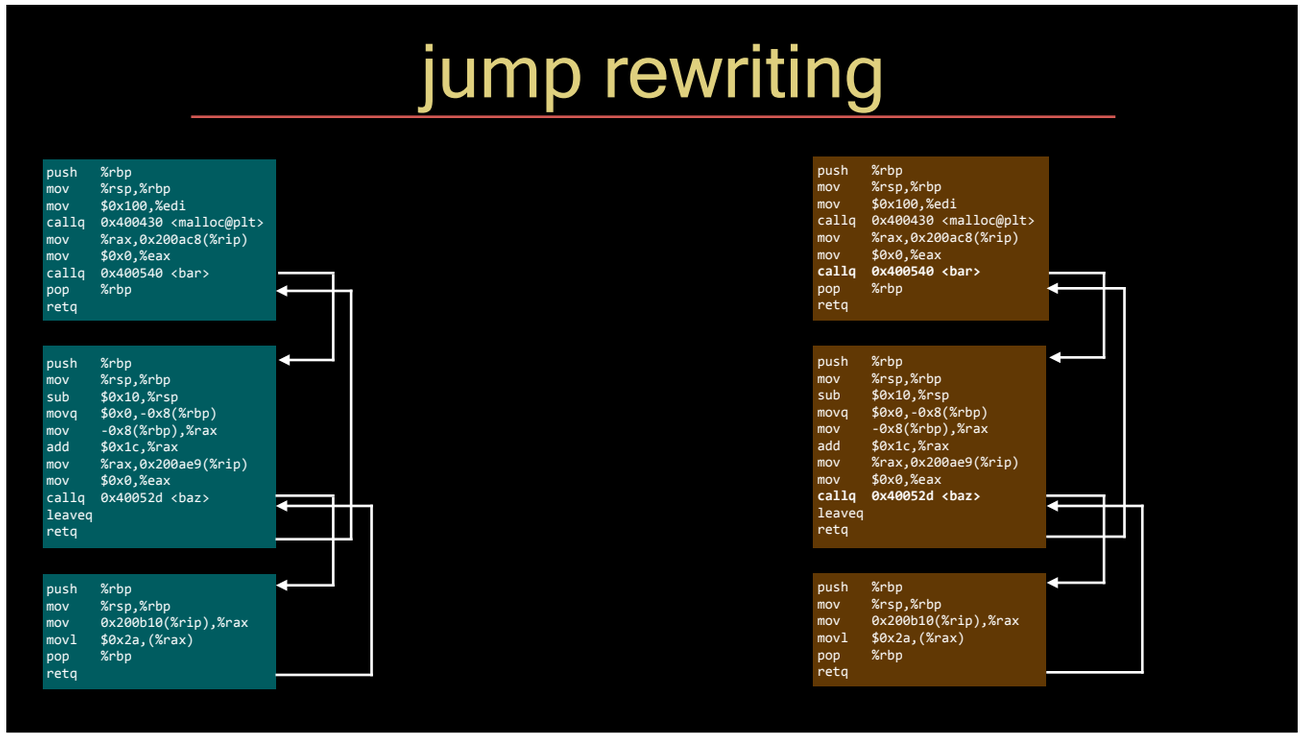
We copy the block and instrument it, but if we don’t patch that control flow change, we could find ourselves jumping back into uninstrumented code! So what we do we <click> take the target address and <click> have it call back into Pin, which would in turn

jump rewriting



compile the jump target’s next code segment, and so on until

jump rewriting



everything has been copied and instrumented. So, I want to point something out here: this process rewrote control flow instructions like calls and jumps, but with the exception of instrumentation code that we might add, most other code stays the same, so for example, memory reads in the original instruction stream will always read the same address as in the instrumented stream.

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

That makes sense when you look at the kind of instrumentation tools that you can write with Pin - most of the examples are either writing events out to a log or incrementing global counters. But, what kind of interesting tools can you build that requires more sophisticated state tracking than this?

Nethercote et al. **Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation** PLDI '07

Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation

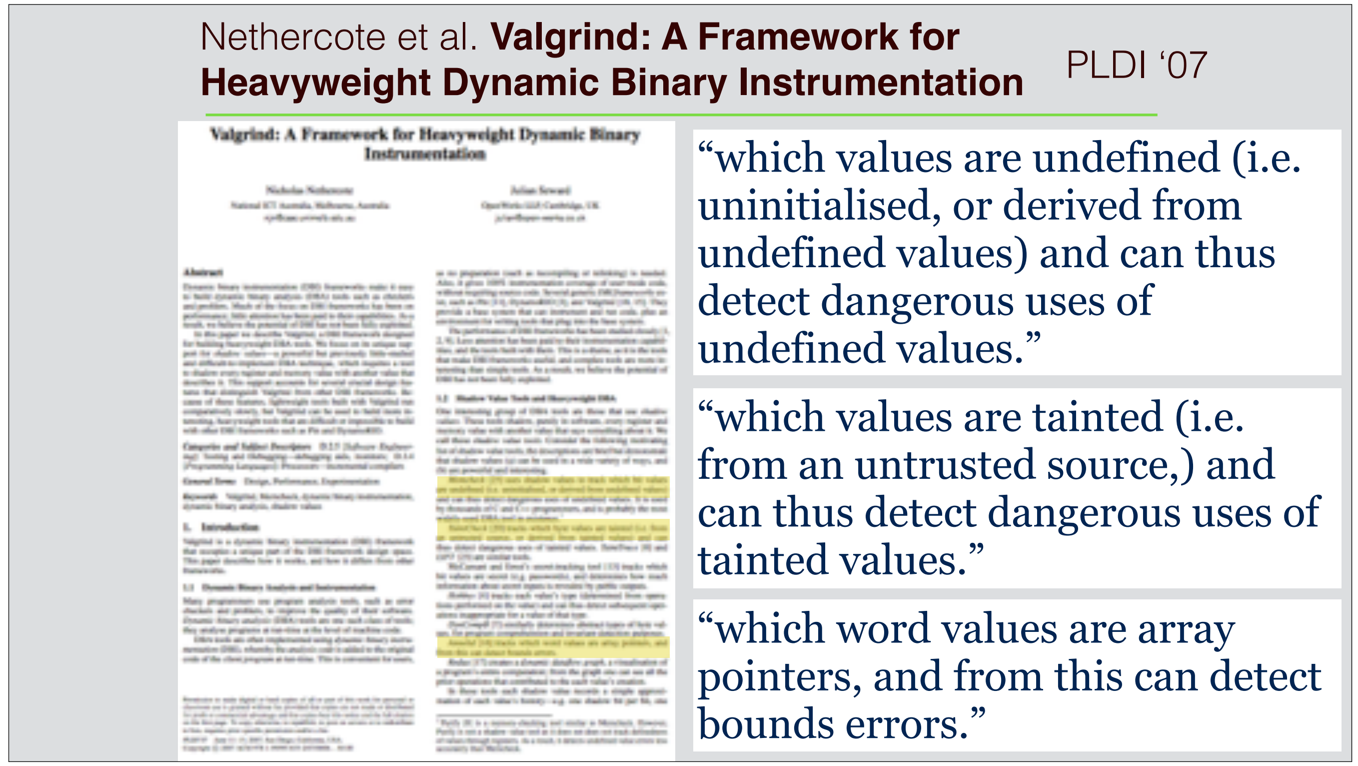
Nicholas Nethercote
National ICT Australia, Melbourne, Australia

Julian Seward
OpenWorks LLP, Cambridge, UK

Abstract

1. Introduction

Some such examples would you couldn't build easily might include tools that track reading memory that you've read but not written to yet, or to annotate how memory is copied through your system, or whether you're walking off the end of an array. What's common about each of these tools is that they require per-memory address state.



Some such examples would you couldn't build easily might include tools that track reading memory that you've read but not written to yet, or to annotate how memory is copied through your system, or whether you're walking off the end of an array. What's common about each of these tools is that they require per-memory address state.

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

Julian Seward
OpenWorks LLP, Cambridge, UK

Abstract

1. Introduction

The primitive that Valgrind, which many of us have used before, uses to make this happen <click> is called shadow memory - it tracks the piece of memory's value and also "interesting facts" about the memory.

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

Julian Seward
OpenWorks LLP, Cambridge, UK

Abstract

1. Introduction

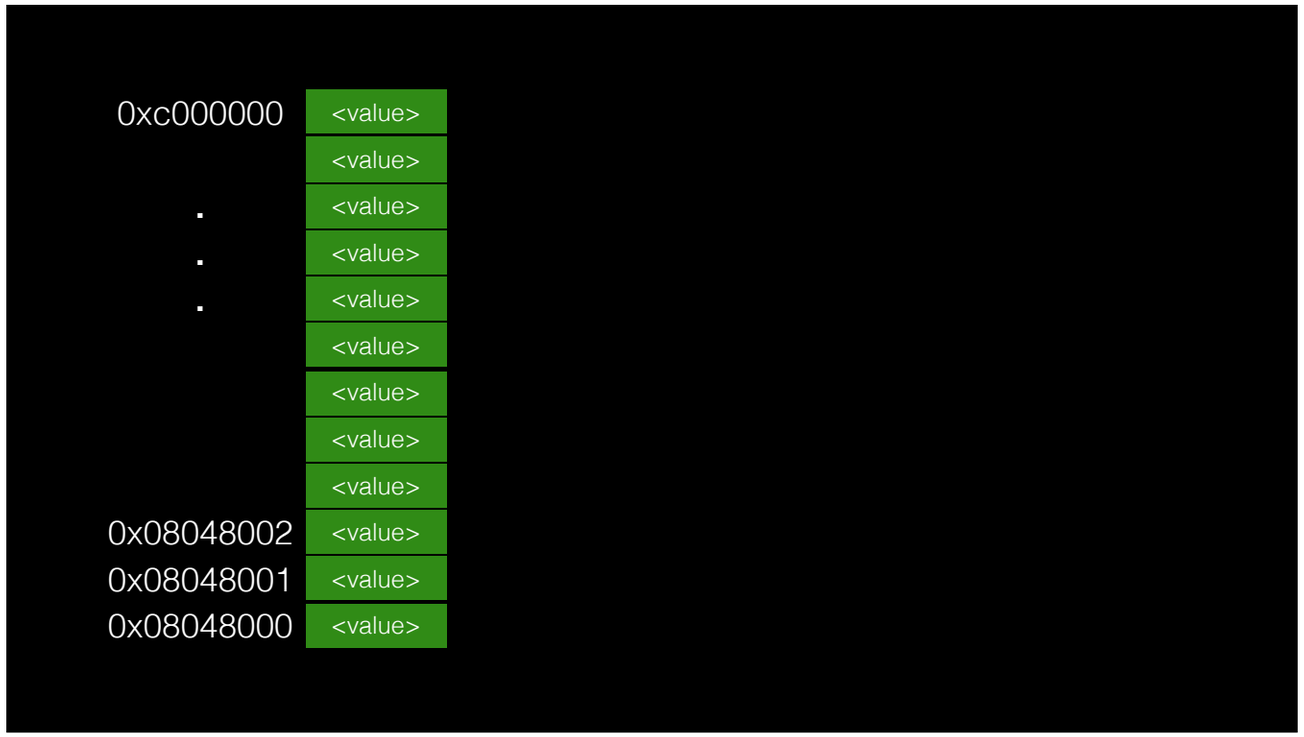
1.1 Dynamic Binary Analysis and Instrumentation

1.2 Shadow Value Tools and Heavyweight DBA

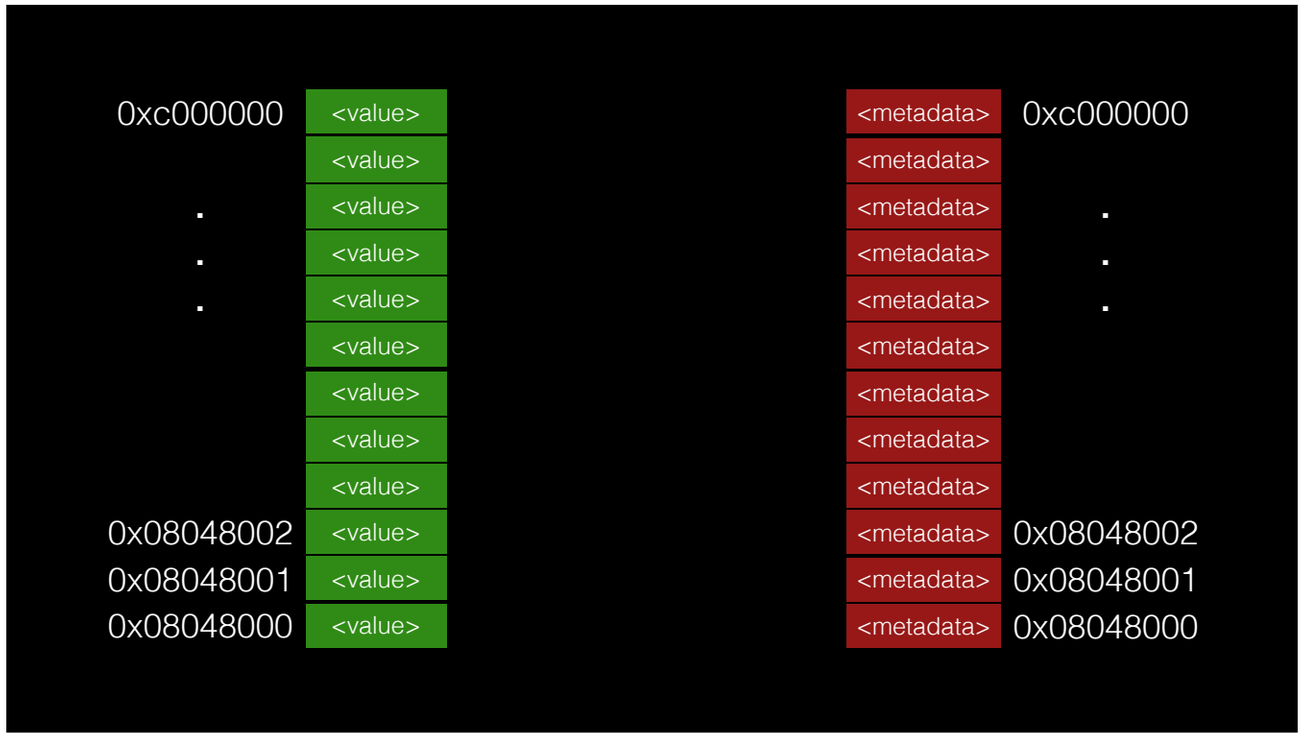
One interesting group of DBA tools are those that use shadow values. These tools shadow, purely in software, every register and memory value with another value that says something about it. We call these shadow value tools. Consider the following motivating list of shadow value tools, the descriptions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

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.

The primitive that Valgrind, which many of us have used before, uses to make this happen <click> is called shadow memory - it tracks the piece of memory's value and also "interesting facts" about the memory.



So if ordinarily memory can be thought of as a big array that spans the addressability of the entire process,



shadow memory can be thought of as associated metadata for each address. What you store in shadow memory depends on what your tool is doing - shadow memory can track a boolean indicating whether a piece of memory has been initialized, or a data structure to store what locks protect that piece of memory, or the size of an array allocated at that address, or anything else you can imagine.

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

Julian Seward
OpenWorks LLP, Cambridge, UK

Abstract

1. Introduction

1.1 Dynamic Binary Analysis and Instrumentation

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.

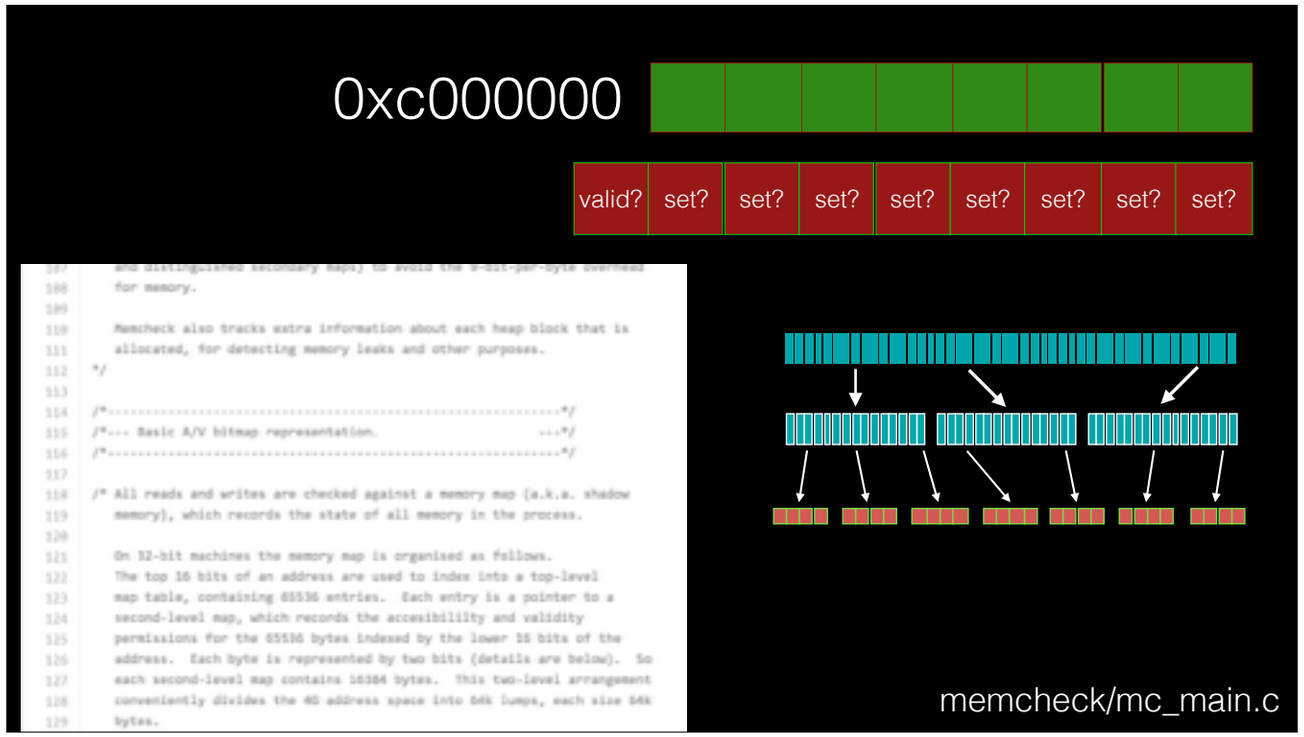
One interesting group of DBA tools are those that use shadow values. These tools shadow, purely in software, every register and memory value with another value that says something about it. We call these shadow value tools. Consider the following motivating list of shadow value tools, the descriptions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

But because Valgrind now needs to rewrite memory access to go to this shadow memory rather as well, these dynamic analysis tools are considerably more heavyweight. Says so right in the paper title.

So what would an example of shadow memory look like? So let’s imagine that we’re building a tool that checks that we have always written to memory before reading from it. This is one of the things that Valgrind’s memcheck plugin does. So memcheck’s shadow state is, conceptually,

for every 8 bit word, another 8 shadow bits corresponding to whether each value bit has been defined, and

one more bit to indicate whether this region of memory can be accessed safely (e.g. is it an allocated part of the heap, or a valid stack frame?). They do some compression on those 9 bits since most of the time all the bits in a byte will be either unset or set.



And then, as the explanation in the source continues, each shadow word is inserted in a sparse two tree keyed on the first half and the last of the value address, not unlike a page table, since most of your process' address space will be the same.

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

Julian Seward
OpenWorks LLP, Cambridge, 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 tools 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 [14], DynamoRIO [3], and Valgrind [16, 17]. 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 [19] 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.

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 [8] and LIFT [18] are similar tools.

McCamant and Ernst's secret-tracking tool [15] tracks which bit values are secret (e.g. passwords), and determines how much information about secret inputs is revealed by public outputs.

Redux [6] 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.

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

Annelid [10] 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 single approximation of each value's history—e.g. one shadow bit per bit, one

---

† Purify [9] is a memory-checking tool similar to Memcheck. However, Purify is not a shadow value tool as it does not track definedness of values through registers. As a result, it detects undefined value errors less accurately than Memcheck.

Val grind is awesome and you should check out both the source code and the paper. So what we’ve seen is dynamic analysis tools being split into two broad camps:

Nethercote et al. **Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation** PLDI '07

Pin was a simpler class of tool which the Valgrind authors call Copy And Annotate: only control flow changes occur and instrumentation happens at the instruction level, as we say. Valgrind is a disassemble and resynthesize tool, where native code gets turned into an intermediary form that tools transform, and then get executed by translating the IR back into native code.

Nethercote et al. **Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation** PLDI '07

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.

Pin was a simpler class of tool which the Valgrind authors call Copy And Annotate: only control flow changes occur and instrumentation happens at the instruction level, as we say. Valgrind is a disassemble and resynthesize tool, where native code gets turned into an intermediary form that tools transform, and then get executed by translating the IR back into native code.

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

PLDI '07

So clearly Valgrind's approach is more computationally expensive, but what do you get for it? Interestingly they point out that <click> while you can build shadow memory tools in Pin, because you can't have an atomic write to both the shadow memory's value and its metadata, multithreading can leave reads in an inconsistent state unless you build up concurrency control yourself. Valgrind takes care of this.

Nethercote et al. **Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation** PLDI '07

"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."

So clearly Valgrind's approach is more computationally expensive, but what do you get for it? Interestingly they point out that <click> while you can build shadow memory tools in Pin, because you can't have an atomic write to both the shadow memory's value and its metadata, multithreading can leave reads in an inconsistent state unless you build up concurrency control yourself. Valgrind takes care of this.

Luk et al. **Pin: Building Customized Program Analysis Tools w/ Dynamic Instrumentation** PLDI ‘05

Unfortunately, though, this comes at a price. If you need the benefits of Shadow Memory you’re going to pay for it; in the Pin paper they point out that Valgrind is about four times slower than Pin.

Luk et al. **Pin: Building Customized Program Analysis Tools w/ Dynamic Instrumentation** PLDI '05

"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."

Unfortunately, though, this comes at a price. If you need the benefits of Shadow Memory you're going to pay for it; in the Pin paper they point out that Valgrind is about four times slower than Pin.

- 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

so we now have enough mechanism that we can talk about building applications that use shadow memory for state tracking, and see how they stack up in practice as opposed to what the theory just tells us.

problem statement

So there’s kind of two questions that we’ve been reaching towards with our discussion about state tracking: for a given piece of memory, what values has it had, and for a given value, where in memory has it ended up? Dynamic analysis sounds like an enticing way to definitively answer these questions, because we can run programs under Pin or Valgrind without needing custom language or compiler support, and one of the claims we’ve made is that there are no false positives. Is that actually true in practice?

problem statement

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

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

So there’s kind of two questions that we’ve been reaching towards with our discussion about state tracking: for a given piece of memory, what values has it had, and for a given value, where in memory has it ended up? Dynamic analysis sounds like an enticing way to definitively answer these questions, because we can run programs under Pin or Valgrind without needing custom language or compiler support, and one of the claims we’ve made is that there are no false positives. Is that actually true in practice?

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,kchris,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

Here’s a paper out of Stanford that watches a system execute in order for developers to learn how sensitive data like passwords propagate through the entire software stack. They do it by annotating, or “tainting”, regions of memory to indicate that that memory’s value originated from a sensitive source.

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,kchris,mendel}@cs.stanford.edu
Stanford University Department of Computer Science

Abstract

1 Introduction

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.

Here’s a paper out of Stanford that watches a system execute in order for developers to learn how sensitive data like passwords propagate through the entire software stack. They do it by annotating, or “tainting”, regions of memory to indicate that that memory’s value originated from a sensitive source.

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,kchris,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

And this isn’t a theoretical paper, they did it with real-world software! They used it to find security bugs in Mozilla, Apache, and Perl.

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,kchris,mendel}@cs.stanford.edu
Stanford University Department of Computer Science

Abstract

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 [8], 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

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.

And this isn't a theoretical paper, they did it with real-world software! They used it to find security bugs in Mozilla, Apache, and Perl.

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

So if I copy a sensitive piece of data verbatim, it makes sense that the copy is just as tainted as the original. But, what about operations that partially operate on tainted data - are the results of those operations also considered tainted?

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

“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.”

So if I copy a sensitive piece of data verbatim, it makes sense that the copy is just as tainted as the original. But, what about operations that partially operate on tainted data - are the results of those operations also considered tainted?

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

So the authors made a particular choice which is to say that any computation that involved tainted memory will have tainted output. They admit this approach is simplistic and could taint too much. Of course, the inverse would be that if a tainted piece of memory is overwritten with something not tainted, the taint mark goes away.

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

> " 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."

So the authors made a particular choice which is to say that any computation that involved tainted memory will have tainted output. They admit this approach is simplistic and could taint too much. Of course, the inverse would be that if a tainted piece of memory is overwritten with something not tainted, the taint mark goes away.

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

This choice also means we can miss things - consider a lookup table where the output is, indeed, a function of the input, but only in the sense that the tainted input affects where in the lookup table to jump to.

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

“ 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.”

This choice also means we can miss things - consider a lookup table where the output is, indeed, a function of the input, but only in the sense that the tainted input affects where in the lookup table to jump to.

methodology

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

So the authors use two features we've talked about - shadow memory to track taint propagating through the system, and executing under an time-traveling debugger in order to replay how the taint propagated.

experiment: mozilla

4.1.1 Mozilla

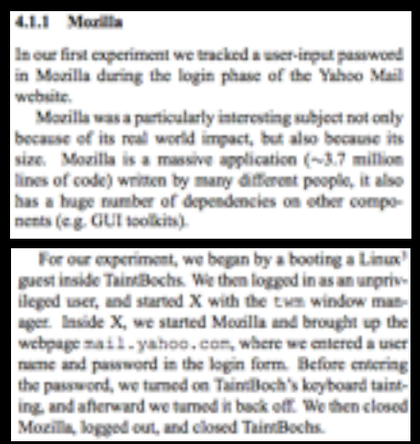
In our first experiment we tracked a user-input password in Mozilla during the login phase of the Yahoo Mail website.

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.

Let's look at one of the experiments that they ran. Here, they entered a password into a form field and watched where the data propagated but was not safely cleared. <click> As it turns out, that password ended up in all sorts of places, starting with the kernel terminal and socket modules, through the window manager queue, through to the browser itself. And all of these modules received the password in plaintext and any buffers stored the plaintext representation of the password. Note that most of these are not in the browser itself but rather in different layers of the stack - without full-system dynamic analysis we wouldn't have learned most of these.

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[1] 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.

Let's look at one of the experiments that they ran. Here, they entered a password into a form field and watched where the data propagated but was not safely cleared. <click> As it turns out, that password ended up in all sorts of places, starting with the kernel terminal and socket modules, through the window manager queue, through to the browser itself. And all of these modules received the password in plaintext and any buffers stored the plaintext representation of the password. Note that most of these are not in the browser itself but rather in different layers of the stack - without full-system dynamic analysis we wouldn't have learned most of these.

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

Herbert Bos
Vrije Universiteit Amsterdam and NICTA*

Abstract

So is taint analysis the best idea since sliced bread? Well, lest you get too terribly excited, other work in the literature has taken a more cynical view, particularly when it comes to trying to analyze malware.

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

They provide a bunch of optimizations but still conclude that there are still serious shortcomings, mostly to do around false negatives.

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

"...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."

They provide a bunch of optimizations but still conclude that there are still serious shortcomings, mostly to do around false negatives.

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

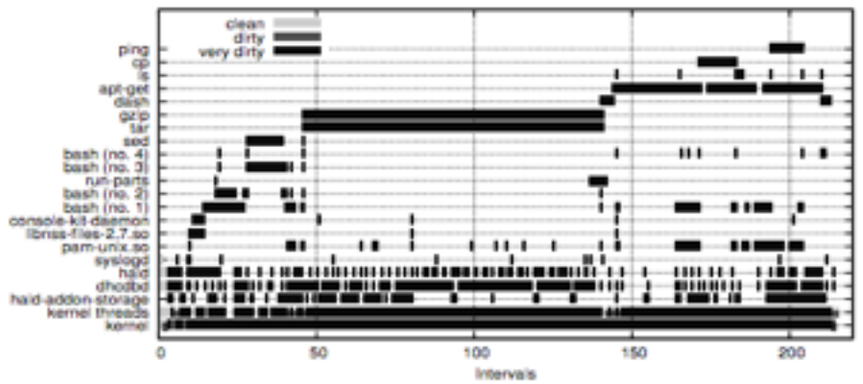


Figure 2. The taintedness of the processes constituting 90% of all context switches. In this and all similar plots the following explanation holds. The x-axis is divided into scheduling intervals, spanning 50 scheduling operations each. Time starts when taint is introduced in the system. In an interval, 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.

If you remember the earlier paper, one of the places that the user's password ended up was in the kernel's random number generator, because it uses keyboard entry as entropy. But, what that means is anything that _reads_ from the random number generator itself becomes tainted, so you end up with this wild propagation of tainted memory through the system - they observe that while trying to analyze a key logger, the whole system's memory essentially becomes "tainted".

so, useful for debugging, for malware analysis, perhaps not so much.

So the second half of this talk is going to be about analyzing CONCURRENT systems. I haven't directly told you this, but everything we've talked about up to this point assumes sequential execution, so processes have one thread running at a time, and virtual machines only have one processor.

King et al. **Debugging Operating Systems with Time-Traveling Virtual Machines** Usenix '05

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

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

So, really, when Sam King and friends back in the time traveling debugging paper talk about nondeterminism, <click> they're really talking about sequential nondeterminism. This means that they're not going to be able to record or replay concurrency bugs.

King et al. **Debugging Operating Systems with Time-Traveling Virtual Machines** Usenix '05

"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

So, really, when Sam King and friends back in the time traveling debugging paper talk about nondeterminism, <click> they're really talking about sequential nondeterminism. This means that they're not going to be able to record or replay concurrency bugs.

problem statement

So there’s kind of two questions that we can ask, and like our problem statement from before they are inverses of each other. How do we solve the same problems we were working on before but with parallelism, and can dynamic analysis help with the difficulty of writing concurrent software?

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?”

So there’s kind of two questions that we can ask, and like our problem statement from before they are inverses of each other. How do we solve the same problems we were working on before but with parallelism, and can dynamic analysis help with the difficulty of writing concurrent software?

concurrent record & replay

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

I’m going to briefly tell you about a few ways that the resarch literature has proposed solving record and replay of a parallel system, and I’m going to argue that none of them are necessarily ideal.

Nethercote et al. **Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation** PLDI ‘07

So I alluded to the problem where writes in tools to shadow memory can become race-ey, because you need to write both the memory value and its shadow value. <click> So Valgrind fixes this by only allowing one thread to run at a time - you can run threaded applications under Valgrind but execution will be serial! So naturally this isn’t appropriate for instrumenting highly-concurrent software running in production.

Nethercote et al. **Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation** PLDI '07

"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."

So I alluded to the problem where writes in tools to shadow memory can become race-ey, because you need to write both the memory value and its shadow value. <click> So Valgrind fixes this by only allowing one thread to run at a time - you can run threaded applications under Valgrind but execution will be serial! So naturally this isn't appropriate for instrumenting highly-concurrent software running in production.

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.

Eraser is a great paper that interposes on locking and unlocking operations to do a slightly different thing - we’ll come back to when we talk about race detectors, but briefly, the approach they take is to

Savage et al: **Eraser: A Dynamic Data Race Detector for Multithreaded Programs** SOSP '97

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.

ACM Transactions on Computer Systems, Vol. 15, No. 4, November 1997.

instrument read and write operations, while keeping track track of what locks are currently being held during those memory operations, and reports any inconsistencies it observes as a data race. This is nice because if your software doesn't have to lock very often, the amount of instrumentation you need to weave in is low and performance won't suffer too much. The downside to looking at things from the perspective of locks is, of course, that the analysis systems is more tightly coupled to the software it's analyzing, so if Eraser is only taught to know about POSIX thread, or Win32 Threads, custom lock implementations would be flagged as false positives.

Savage et al: **Eraser: A Dynamic Data Race Detector for Multithreaded Programs** SOSP '97

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"

instrument read and write operations, while keeping track track of what locks are currently being held during those memory operations, and reports any inconsistencies it observes as a data race. This is nice because if your software doesn't have to lock very often, the amount of instrumentation you need to weave in is low and performance won't suffer too much. The downside to looking at things from the perspective of locks is, of course, that the analysis systems is more tightly coupled to the software it's analyzing, so if Eraser is only taught to know about POSIX thread, or Win32 Threads, custom lock implementations would be flagged as false positives.

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

VEE ‘08

Execution Replay for Multiprocessor Virtual Machines

A common trick in systems building is to leverage existing hardware for new purposes - to this end, Dunlap and his colleagues implemented a way of recording and replaying a multicore system’s shared memory accesses by exploiting the virtual memory hardware present in all our computers today.

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

VEE ‘08

So in order to do this, they implement a protocol called CREW, which stands for concurrent read / exclusive write. What this means is that a region of memory is either readable by everyone but writable by nobody, or writable by exactly one processor and readable by nobody.

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

VEE ‘08

- **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.”

So in order to do this, they implement a protocol called CREW, which stands for concurrent read / exclusive write. What this means is that a region of memory is either readable by everyone but writable by nobody, or writable by exactly one processor and readable by nobody.

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

VEE ‘08

So they built their system in a virtual machine monitor, Xen, that runs underneath the OS and virtualizes the OS’ virtual memory mappings.

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

VEE ‘08

“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.”

So they built their system in a virtual machine monitor, Xen, that runs underneath the OS and virtualizes the OS’ virtual memory mappings.

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

VEE ‘08

- **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.”

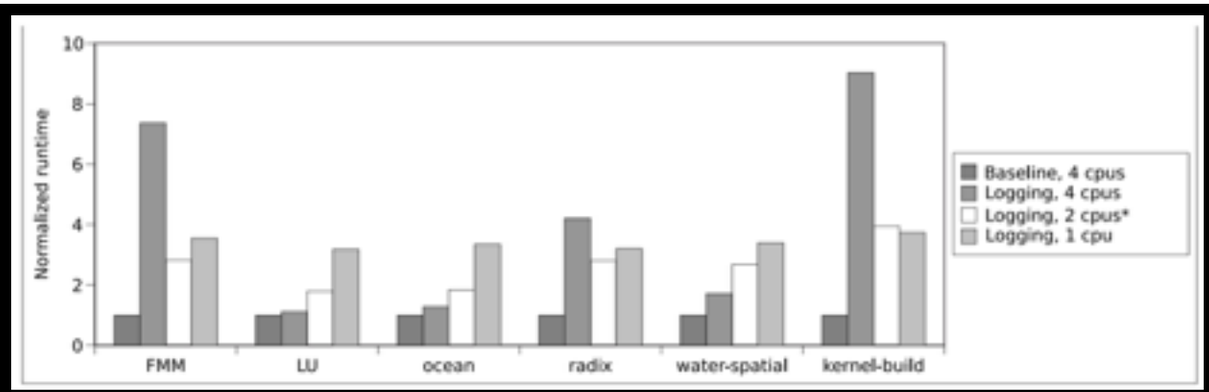
The idea is that if you are in concurrent read mode and someone tries to write, you trap into the hypervisor and you switch to exclusive write mode by revoking read access at the MMU level. You record that this mode switch took place, and will use that mode switch in order to enforce the same order of events during replay. We’ll talk a bit about the theory behind this in a bit.

CREW limitations

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

There are a few limitations here - using existing hardware techniques is a great idea but they operate on page level rather than byte level. Pages are generally 4k in size. As a result, there might be CREW events generated for two unshared pieces of memory if they still reside on the same memory page, which doesn't affect correctness but can force ultimately unnecessary synchronization

DMA, which is where hardware peripherals like network cards move data around without being coordinated by the processor, was listed as a problem because those devices end up operating outside the dynamic analysis system's walled garden, but virtualizing hardware devices is now possible.



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.

Replaying a 4 CPU parallel workload can incur overhead up to 10x, depending on how much data is shared for a given workload. Also, the execution logs can take up a tremendous amount of space. FMM, Fast Multipole Method, logs at upwards of 100GB/day.

Race detection

Hopefully I’ve convinced you that full-system record and replay is still not super feasible in general. The last topic of my presentation is therefore about a simpler but very practical problem - rather than recording enough state in the system to play it back fully, we just need to observe enough of the system to report when a data race occurs.

race detection techiques

- Lockset algorithm
- Happens-before relation
- Hybrid techniques

So there are two major techniques that I'll tell you about today, and the third one will be a case study of a real-world production race detector that turns out to be a combination of the first two.

```
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

What does a data race look like? Well, the simplest one that I can think of is a shared global variable that is modified by two threads without some sort of mutual exclusion technique like atomics or locks.

```
// 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

So protecting a global variable in a multi-threaded program is probably pretty clear, but you can have the same kind of bug even if the variable's scope is local - here, the reference count isn't protected so you can potentially leak memory or cause a double-delete.

```
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

A more interesting data race that I like to point out comes from compiler optimizations - because the compiler only knows about single-threaded execution

```
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

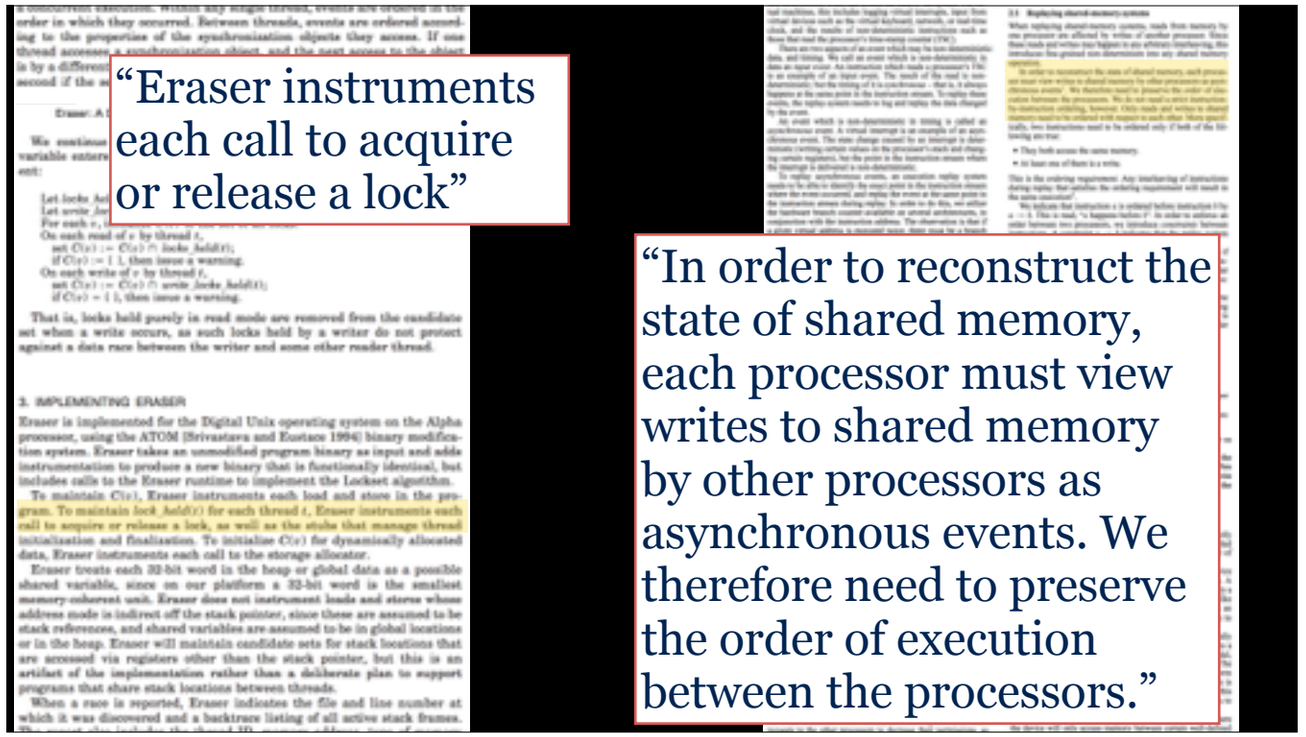
since there’s no relationship between “done” and any other statement in the Thread2 function, that operation can be reordered by the compiler arbitrarily, like so

```
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

So, this means that Thread1 will think Thread2 has completed its useful work before it really has.

So a straightforward way to implement a race detector would be to instrument shared reads and writes, and cross-reference those memory operations with the locks that are currently held. By this point in the talk, you have some idea about how to implement such a tool!



So a straightforward way to implement a race detector would be to instrument shared reads and writes, and cross-reference those memory operations with the locks that are currently held. By this point in the talk, you have some idea about how to implement such a tool!

Savage et al: **Eraser: A Dynamic Data Race Detector for Multithreaded Programs** SOSP '97

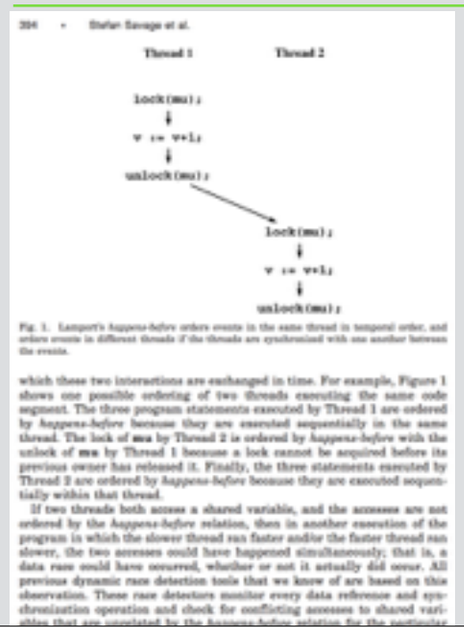


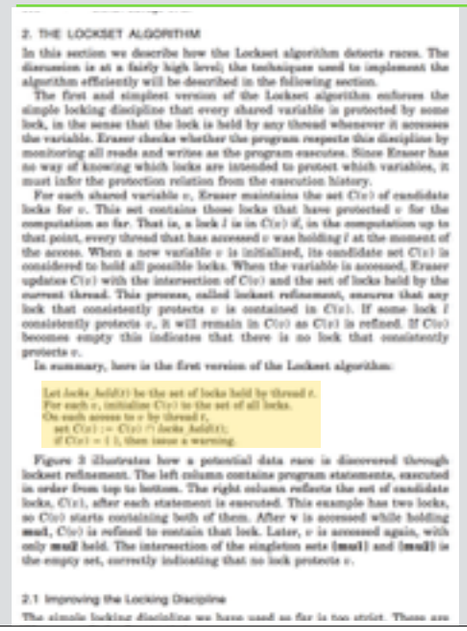
Fig. 1. Lamport's *happens-before* orders events in the same thread in temporal order, and orders events in different threads if the threads are synchronized with one another between the events.

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

So the key idea behind Eraser is that it wants to ensure there is at least one lock that consistently protects a shared resource. So in this diagram, threads 1 and 2 correctly protect access to v because they use the mu lock in both cases. If the threads used different locks, then there's no lock that would consistently protect v, so Eraser would report this.

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) ∩ locks_held(t);
    if C(v) = { }, then issue a warning.
```

Figure 2 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

Instead, the authors keep track of the set of locks that have protected a piece of memory, and every time that memory is accessed they intersect the set of currently-held locks with the previously-observed lockset. If the intersection is empty, then there is no lock that consistently protects that memory in our execution!

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 3:16 *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.

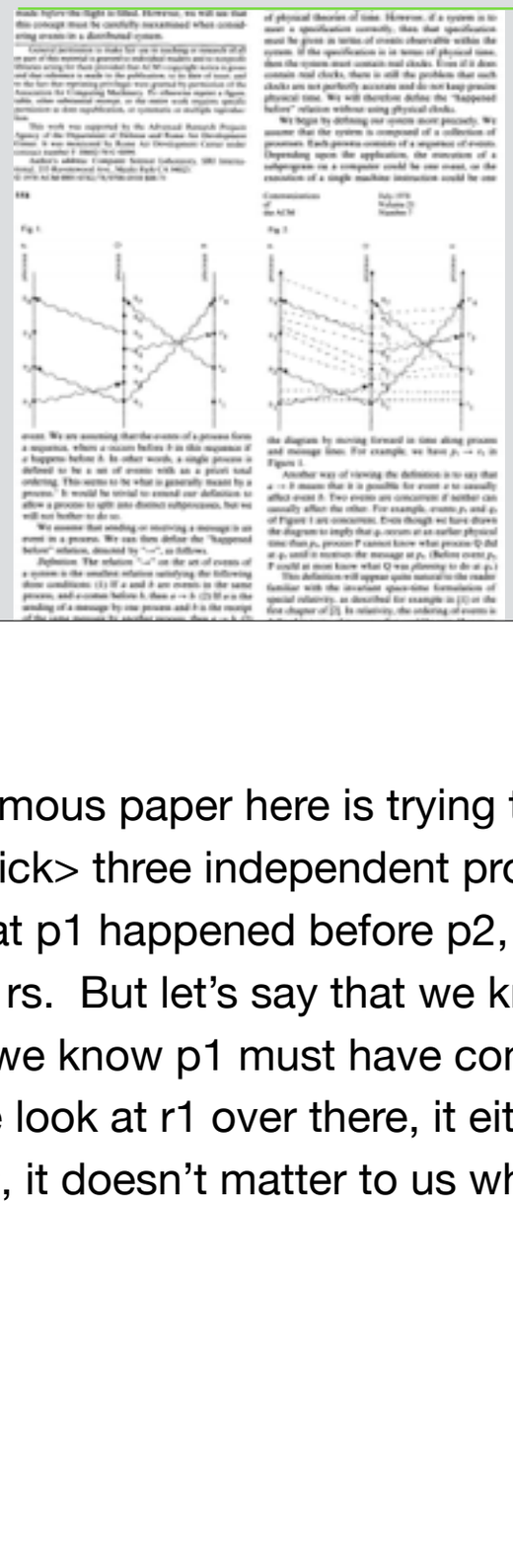
In this paper, we discuss the partial ordering defined by the "happened before" relation, and give a distributed algorithm for extending it to a consistent total ordering of all the events. This algorithm can provide a useful mechanism for implementing a distributed system. We illustrate its use with 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

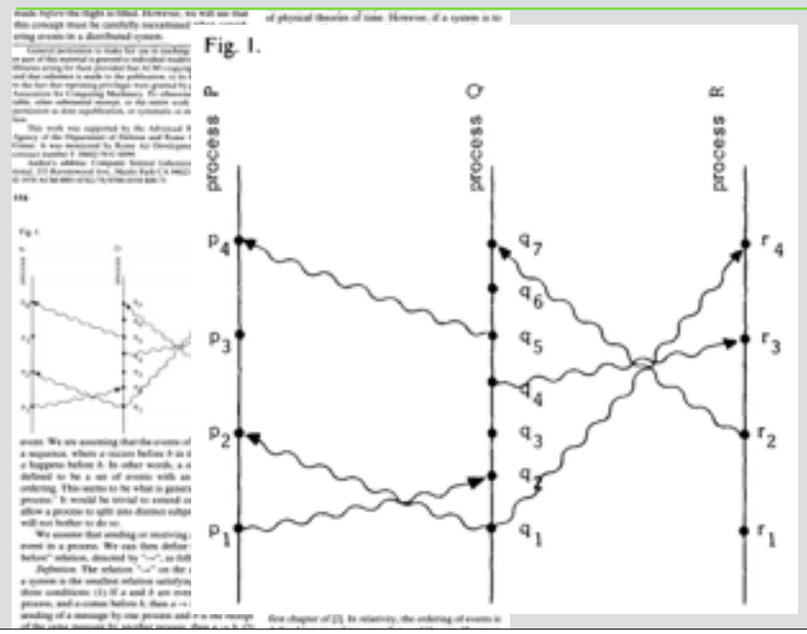
The second technique, which uses a property called the “happens-before” relation, borrows from the distributed systems literature. After all, what is a multicore computer but a distributed system in a tiny box? If you wanted to order the events in a concurrent system, you might think of using wall clock time as the objective arbiter of what happened first. But this turns out to be impractical - synchronizing wall clock time across a distributed system is really hard to do.

Lamport et al: **Time, Locks and the Ordering of Events in a Distributed System** CACM ‘78



So what Lamport wrote about in his famous paper here is trying to understand the relationship of events that are causally related to each other. So there are <click> three independent processes P,Q,R and you read this diagram from the bottom up, the y axis is “time” in the sense that p1 happened before p2, which happened before p3, but we might not know where they lie in time relative to the qs or the rs. But let’s say that we know that operation q2 depended on the p1 happening - they’re causally related. In that case, we know p1 must have come before q2, so in the diagram we have an edge connecting the two. By contrast, if we look at r1 over there, it either happened before, during, or after p1, but because its result doesn’t affect any other process, it doesn’t matter to us when it happened, only that it happened before r2, which relates to q7!

Lamport et al: **Time, Locks and the Ordering of Events in a Distributed System** CACM ‘78



So what Lamport wrote about in his famous paper here is trying to understand the relationship of events that are causally related to each other. So there are <click> three independent processes P,Q,R and you read this diagram from the bottom up, the y axis is “time” in the sense that p1 happened before p2, which happened before p3, but we might not know where they lie in time relative to the qs or the rs. But let’s say that we know that operation q2 depended on the p1 happening - they’re causally related. In that case, we know p1 must have come before q2, so in the diagram we have an edge connecting the two. By contrast, if we look at r1 over there, it either happened before, during, or after p1, but because its result doesn’t affect any other process, it doesn’t matter to us when it happened, only that it happened before r2, which relates to q7!

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

VEE ‘08

Execution Replay for Multiprocessor Virtual Machines

So what does this have to do with parallel record and replay systems? Well, Dunlap and his colleagues observed

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

VEE ‘08

that they can apply the same techniques to solve the multiprocessor record and replay problem - rather than recording the exact and objective timing for every memory operation, they only track the relative ordering of shared memory accesses, which they get from their CREW protocol. Therefore, replaying the execution of the full system reduces to replaying the right order of CREW events.

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

VEE ‘08

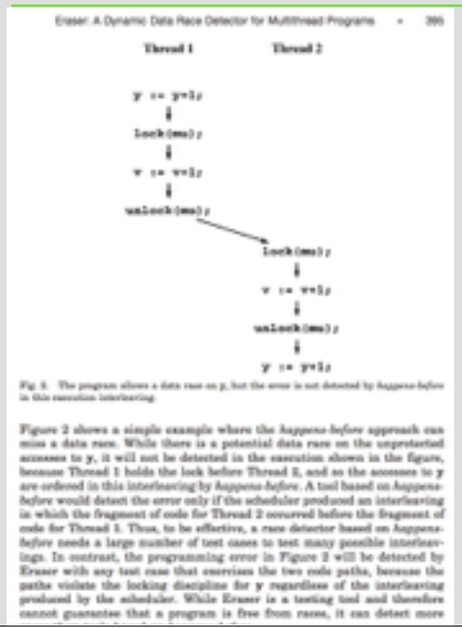
“In order to reconstruct the state of shared memory, each processor must view writes to shared memory by other processors as asynchronous events. We therefore need to preserve the order of execution between the processors.”

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

that they can apply the same techniques to solve the multiprocessor record and replay problem - rather than recording the exact and objective timing for every memory operation, they only track the relative ordering of shared memory accesses, which they get from their CREW protocol. Therefore, replaying the execution of the full system reduces to replaying the right order of CREW events.

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

SOSP ‘97



Eraser: A Dynamic Data Race Detector for Multithread Programs · 395

Fig. 2. The program shows 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

The Eraser authors argue that their method is superior to using happens-before can miss - there’s _implicit_ happens-before between thread one and thread two here, since there’s a synchronization event between accesses to y, but y still is outside the critical section, so the scheduler would have to run these threads in a different order for happens-before to observe the race. With dynamic lockset tracking, we would be guaranteed to observe that no locks are protecting Y.

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

PPoPP '03

Another interesting limitation that leads into the final detector we'll study is that if your concurrency model isn't entirely thread-based then the lockset approach isn't appropriate. This paper from '03 alludes to the Communicating sequential processes programming model which is what Golang uses.

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

PPoPP '03

"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."

Another interesting limitation that leads into the final detector we'll study is that if your concurrency model isn't entirely thread-based then the lockset approach isn't appropriate. This paper from '03 alludes to the Communicating sequential processes programming model which is what Golang uses.

- 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

So we looked at Lockset, which computes this intersection operation for every shared memory operation, which is great if you're only using traditional locks. Then we looked at this happens-before relation which builds this causality graph, which is more flexible but is dependent on execution order and actually ends up being slower to run.

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
0 Institutskii per
Dolgoprudny, 141700, Russia
timur.iskhodzhanov@phystech.edu

ABSTRACT

Categories and Subject Descriptors

General Terms

Keywords

1. INTRODUCTION

So both techniques on their own run pretty slowly but actually a neat thing is that you can combine the two. ThreadSanitizer is a production-caliber race detector written by some folks at Google. This early paper covers an implementation written in Valgrind, but has since been implemented as a part of GCC & LLVM, and Go’s race detector is built on top of this rewrite.

a hybrid approach

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

briefly the idea is that you use happens-before to reduce the amount of false positives but keep the efficient lockset approach. A traditional happens-before would have to track every memory access, not synchronization operation , so we make it a bit sloppier here but reduce the complexity of the happens-before graph that we have to build.

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

Let's wrap up by looking at how well this technique works - they compared it against Valgrind's memory checker which doesn't do race detection but performs similar instrumentation, and their benchmarks were close to each other. On large tests (e.g. unit), ThreadSanitizer can be twice as fast as Helgrind, Valgrind's built-in race detector. The memory consumption is also comparable to Memcheck and Helgrind.

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.

And, probably the greatest success story I can give you comes from the end of the paper, where the tool finds a race condition that had eluded the Chromium team for months … on the first run.

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

Let’s look at what we’ve covered in this talk

…and we used these to consider a particular usecase, race detection of concurrent software. These components all work in the service of being able to record, analyze, and replay execution without sacrificing performance.

the trace is out there

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

I want to leave you with a few things in this space that we didn't have time to talk about, and things that are very much unsolved by both practiioners and researchers - first of all, there's lots of interesting work on the hardware side of things, both in terms of repurposing existing hardware to make analysis faster, such as the x86 MMU that we saw in this talk, Intel's new Processor Trace functionality, or the Itanium's advanced load address table, or designing entirely new hardware to make these sorts of tools run fast.

You can also imagine treating these execution traces as big data problems. What is the right way to mine huge trace logs for bugs? What is the right querying interface for execution data miners?

Lastly, we've only talked about text-based analysis, but there is some really cool work in visualizing the output from dynamic analysis tools to make it easier for developers to understand how their program is running.

Thanks

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

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