

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



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

The Concept of Dynamic Analysis

Thomas Ball

Bell Laboratories
Lucent Technologies
tbl@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 frequency of program events in a code 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 dominators, postconditions, and regions. Comparison of these dynamically computed relationships to their static counterparts can point to areas of code requiring more testing and can aid programmers in understanding how a program and its test sets relate to one another.

1 Introduction

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

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

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

dynamic analysis, 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.

The Concept of Dynamic Analysis

Thomas Ball

Bell Laboratories
Lucent Technologies
tbl@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 frequency of program execution in a code section 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 dominators, postconditions, and regions. Comparison of these dynamically computed relationships to their static counterparts can point to areas of code requiring more testing and can aid programmers in understanding how a program and its test sets relate to one another.

1. Introduction

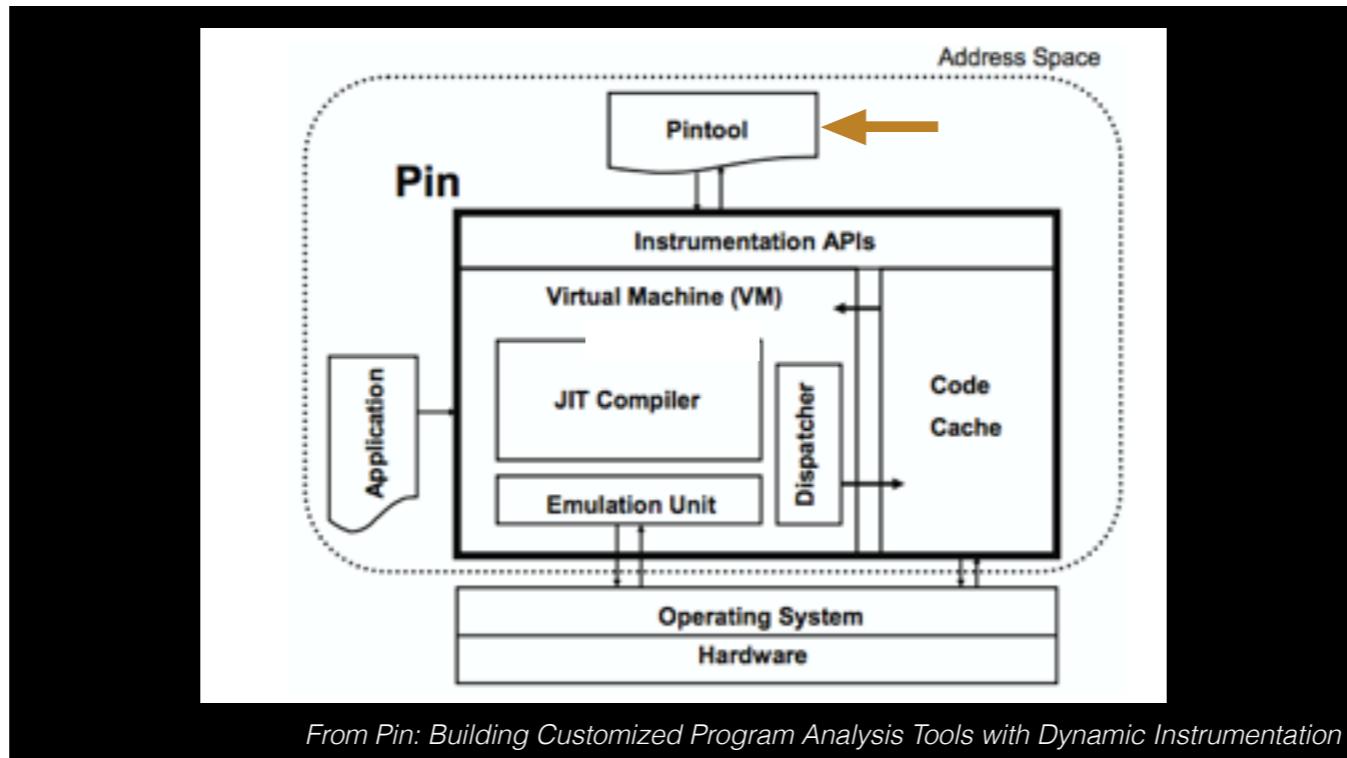
Dynamic analysis is the analysis of the properties of a running program. In contrast to static analysis, which examines a program's text-to derive properties that hold for all executions, dynamic analysis derives properties that hold for one or more executions by examination of the running program (usually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

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

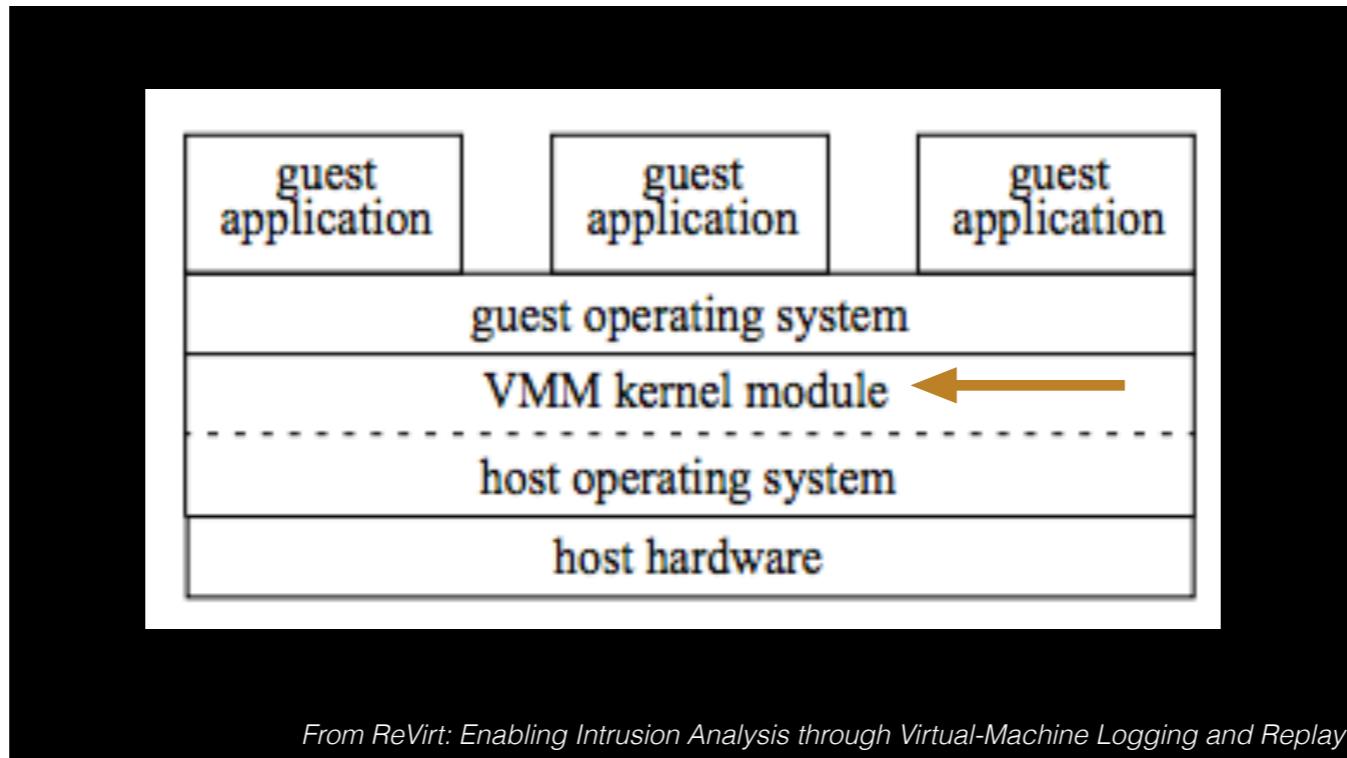
- *Precision of information*: dynamic analysis typically involves instrumenting a program to examine or record certain aspects of its run-time state. This instrumentation can be tuned to collect precisely the information needed

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

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.



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

- ➊ Huge gulf between what tools practitioners use in industry, and academic dynamic analysis work
- ➋ Many techniques shared with other cool areas like OS virtualization and compiler theory
- ➌ 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.

dynamic vs static analysis

- Static analysis is a conservative approximation of runtime behavior

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.

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

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.

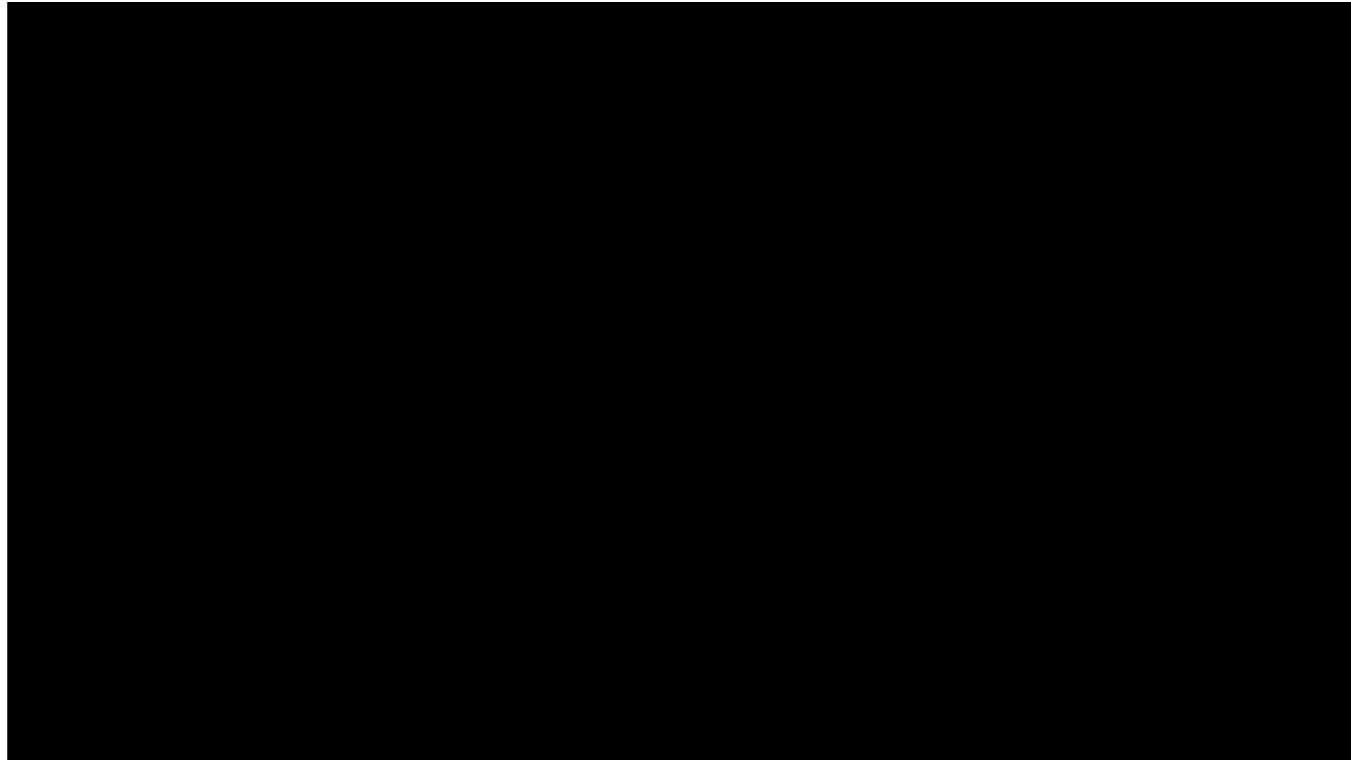
dynamic vs static analysis

- 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!

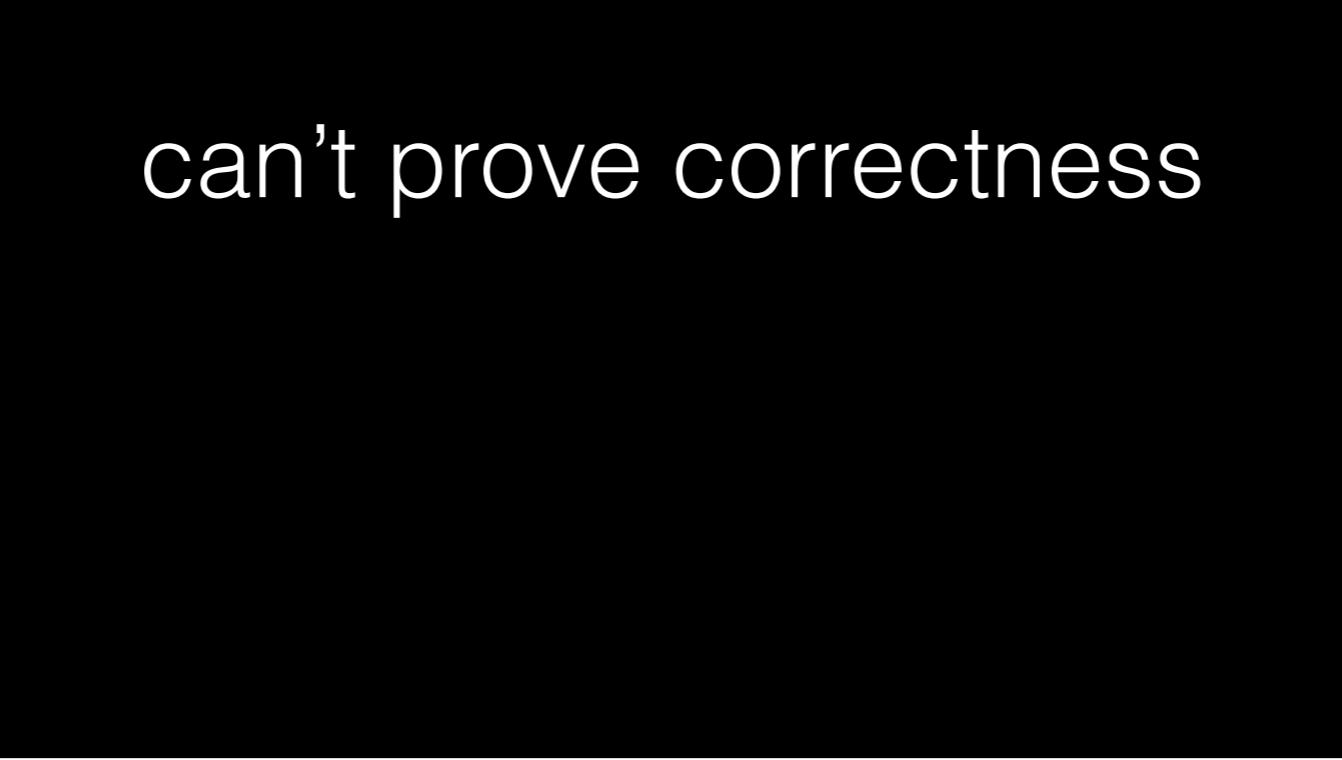
I think it's worth taking a moment to contrast dynamic analysis with static techniques like formal verification, which Caitie gave a great talk about earlier today, and language features like type systems. <click>

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 sophisticated 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.

```
→ /tmp gcc foo.c
```

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=0x7fffffff438) 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

J. Mickens: The Night Watch

login Nov 2013

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=0x7fffffff438)
(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-Traveling Debugging



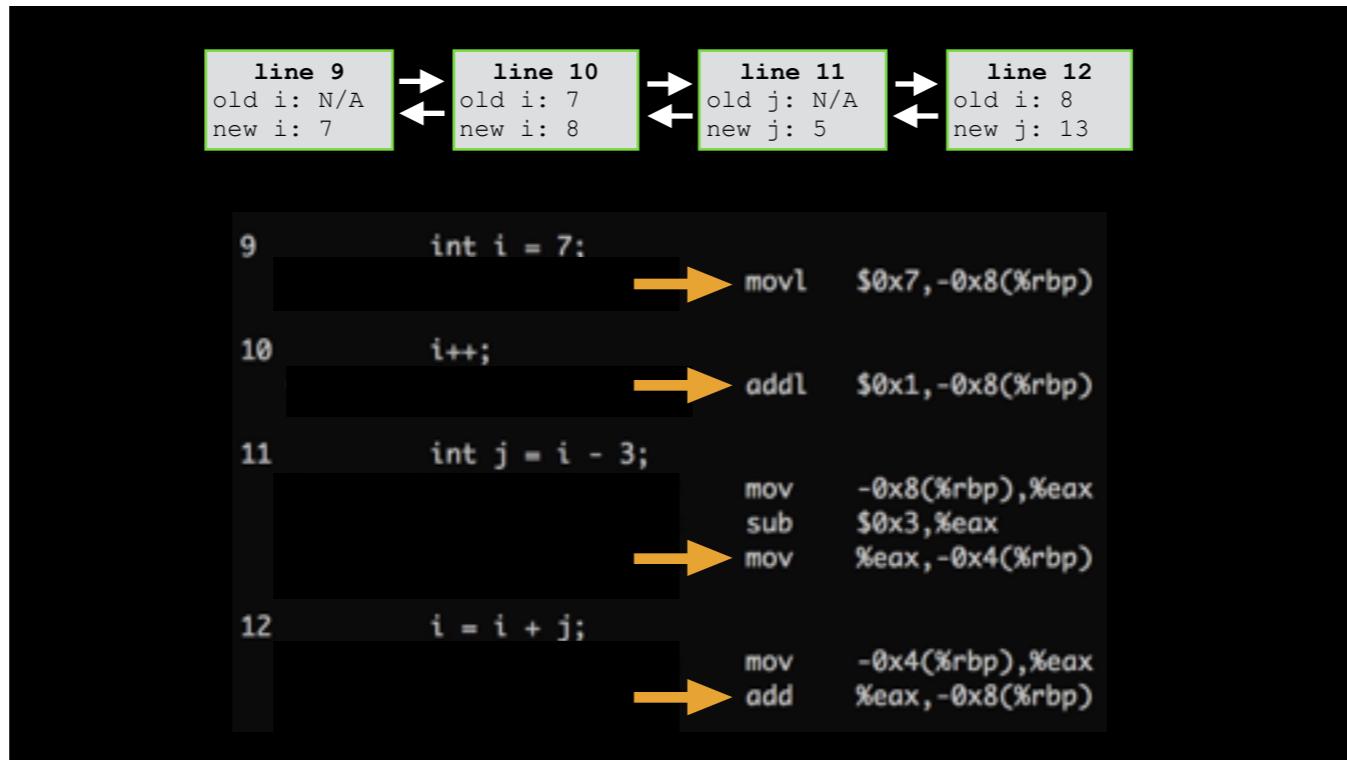
time travelling debugger to do this. At the highest level, what distinguishes an ordinary debugger from an time-traveling one is that the latter is able to reconstruct program state at any point in a process' execution. If we had an time-traveling debugger we could see where our pointer got overwritten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

```
8 void bar() {  
9     int i = 7;  
10    i++;  
11    int j = i - 3;  
12    i = i + j;  
  
9     int i = 7:           → movl $0x7,-0x8(%rbp)  
10    i++;                → addl $0x1,-0x8(%rbp)  
11    int j = i - 3;       → mov -0x8(%rbp),%eax  
                           sub $0x3,%eax  
                           mov %eax,-0x4(%rbp)  
12    i = i + j;          → mov -0x4(%rbp),%eax  
                           add %eax,-0x8(%rbp)
```

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

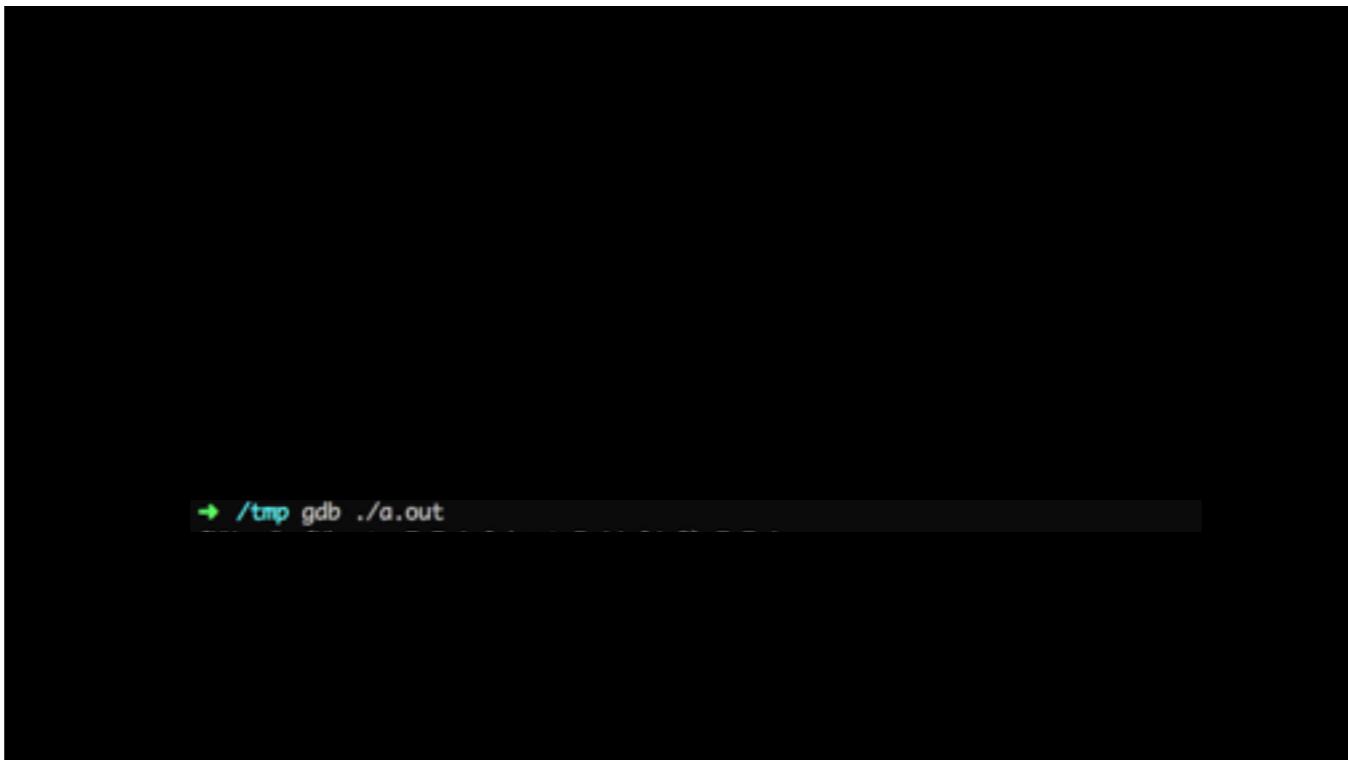


stored a doubly linked list of all the writes to memory that we've observed for a program run with their before and after value, then we can see future state of the program by reading the list forwards and past state by reading it backwards.

```
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

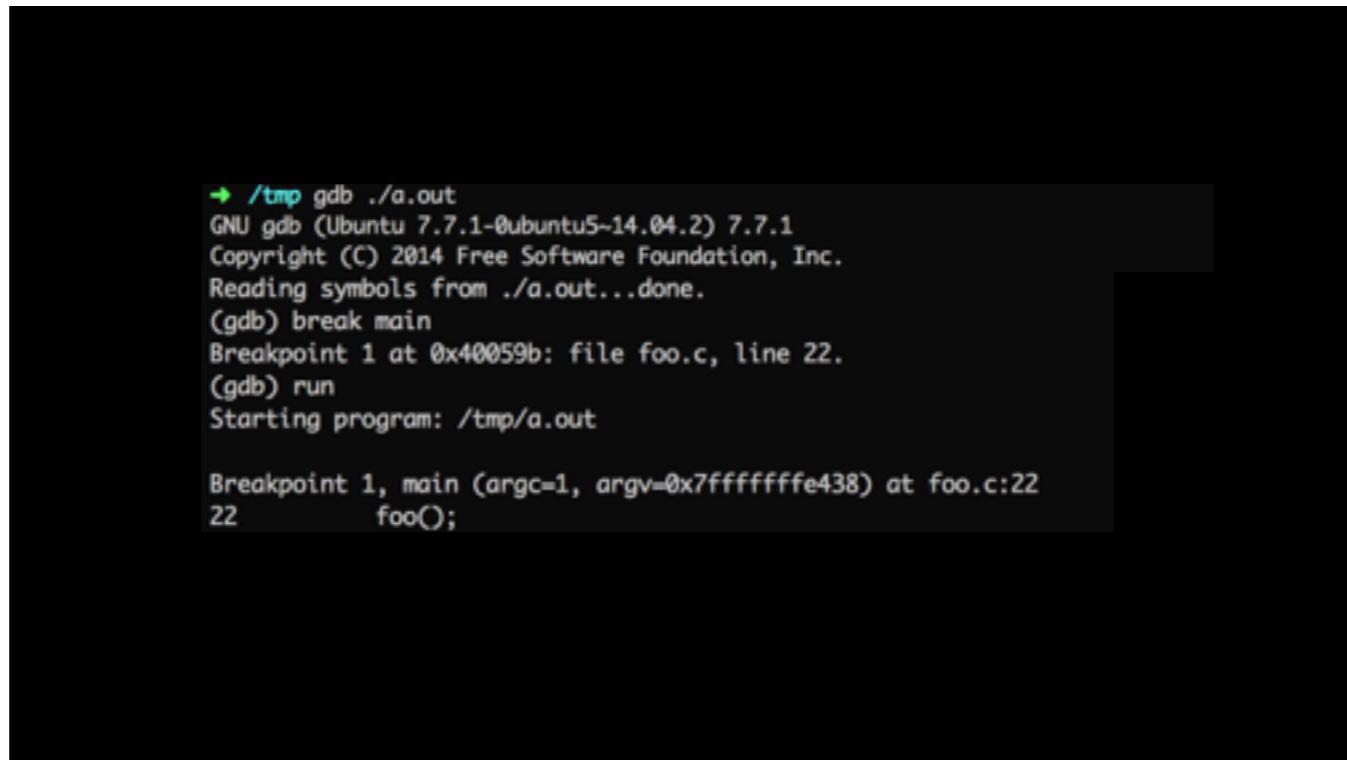
As it turns out, some of you may be aware that GDB, the popular open source debugger for C languages, actually supports time-traveling debugging, and this is exactly how it's implemented under the hood! When you start a time-traveling recording session, every register or memory change that the debugger 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,

A screenshot of a terminal window showing a GDB session. The session starts with the command `gdb ./a.out`, followed by setting a breakpoint at `main` and running the program. The output shows the program has crashed at line 22 of `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.
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=0x7fffffff438) at foo.c:22
22      foo();
```

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=0x7fffffff438) 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,

```
Copyright (C) 2014 Free Software Foundation, Inc.  
Reading symbols from ./a.out...done.  
(gdb) break main  
Breakpoint 1 at 0x400059b: file foo.c, line 22.  
(gdb) run  
Starting program: /tmp/a.out  
  
Breakpoint 1, main (argc=1, argv=0x7fffffff438) at foo.c:22  
22      foo();  
(gdb) record  
(gdb) continue  
Continuing.  
  
[process 4866] #1 stopped.  
0x0000000000400538 in baz () at foo.c:7  
7      *ptr = 42;
```

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,

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

Breakpoint 1, main (argc=1, argv=0x7fffffff438) at foo.c:22
22      foo();
(gdb) record
(gdb) continue
Continuing.

[process 4866] #1 stopped.
0x0000000000400538 in baz () at foo.c:7
7      *ptr = 42;
(gdb) watch ptr
Hardware watchpoint 2: ptr
(gdb) set exec-direction reverse
(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,

```
(gdb) continue
Continuing.

[process 4866] #1 stopped.
0x0000000000400538 in baz () at foo.c:7
7          *ptr = 42;
(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;
```

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=0x7fffffff438) 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,

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

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

Stéphane Ducasse
LISTIC
Université de Savoie, France
www.listic.ensr.fr/~ducasse

Abstract:

Traditional debugging and stepping execution trace are well accepted techniques to understand deep problems 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, as 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 UNTECK 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 [DEN02], 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 reacquire the exact same context [LJ99].

The questions a programmer has are often: "where was this variable set?", "why is this object reference null?" 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 Omnicient Debugger is a first attempt to answer these problems [Lawill], 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.

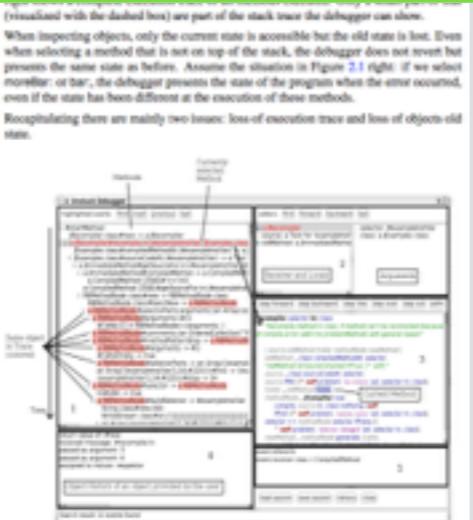


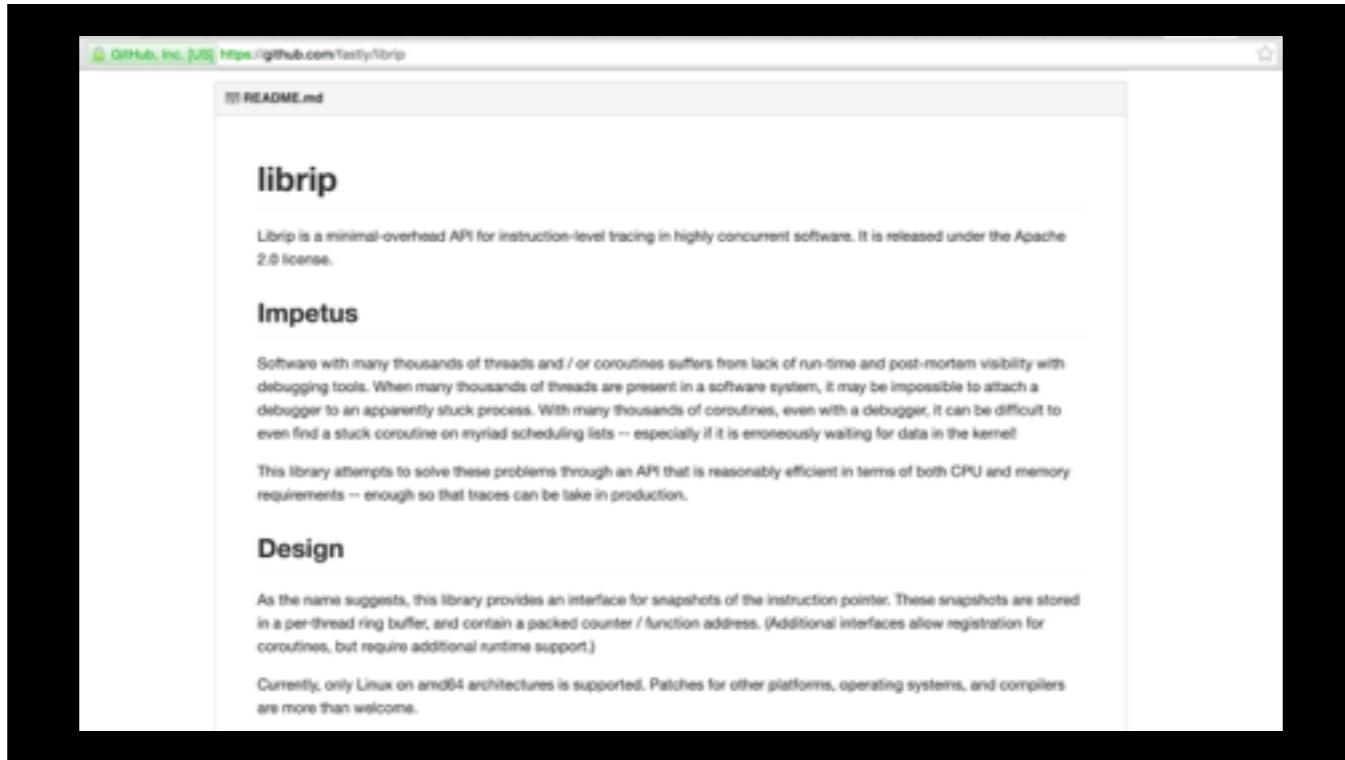
Figure 2: The user interface of the Unstack Debugger.

changed. We can step through these highlighted methods in the method trace, or use the stopping 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 over the context menu in the source view. Right: the result of the actions 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.

Efficient time-traveling



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.

Debugging operating systems with time-traveling virtual machines

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

Abstract

Operating systems are difficult to debug with traditional cyclic debugging. They are non-deterministic; they can run for long periods of time before they interact directly with hardware devices. This makes it difficult to reason about the state of the system by the end 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 process. We also employ time-slicing to support a debugger to debug an OS in-use, implementing commands such as resume breakpoint, resume watchdog, and resume single step. The space and time overheads needed to support time travel are reasonable for the target operating system. Measurements show the value of our time-traveling virtual machine by using it to understand a few several OS bugs that are difficult to find with standard debugging tools. Because debugging is especially helpful in finding bugs that are fragile due to non-determinism, bugs in device drivers, bugs in the kernel, and bugs in memory that corrupt stack frames are presented.

I. Introduction

Computer programmers are all-too-familiar with the task of debugging complex software through a combination of detective work, paperwork, and systematic search. Tracking down a bug usually starts with narrowing down to an error in the program's logic or memory usage as a "bug". The programmer¹ then sets to start from the final 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 source. Debuggers implement one or more logical or memory mechanisms to examine the state of the program at a given point in its execution. Armed with

¹In this paper, "programmer" refers to a person who is writing the application code. "User" refers to the person who is using the application.

this information, the programmer then re-runs the program, steps it at or near points in its execution history, observes the state at this point, and repeats.

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

First, the act of debugging may perturb the state of the operating system. The programmer is often a time-sharing 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. In addition, the programmer may interact with the OS via memory-mapped I/O. The memory-mapped I/O interface allows the programmer to interact with hardware devices directly. The memory-mapped I/O interface is a basic functionality in the debugged OS, such as reading and writing memory locations, setting and handling breakpoints, and communicating with the source debugger (e.g., through the serial line). Unfortunately, basic functionality may be implemented in the OS. In addition, some hardware devices, such as disk drives, may interact with the OS via memory-mapped I/O to access hardware devices, and this functionality may not work on a real OS.

Finally, operating systems interact directly with hardware devices. Devices are sources of non-determinism that hinder cyclic debugging: they return data and generate events at unpredictable, irregular intervals. Devices may also fail due to transient errors if a programmer makes changes 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. Duran, and Peter M. Chen
University of Michigan

Abstract

Operating systems are difficult to debug with traditional cyclic debugging. They are non-deterministic; they can run for long periods of time before they interact directly with hardware devices. They are also non-transparent to the user, which makes them difficult to debug. 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 process, without any explicit support from the host operating system. We show how to use a time-traveling debugger to enable a programmer to debug an OS in its native environment, implementing commands such as resume breakpoint, resume watchdog, and reverse single step. The space and time overheads needed to support time travel are reasonable for the target operating system. The authors believe that the value of this time-traveling virtual machine lies in its understandability and its several OS bugs that are difficult to find with standard debugging tools. Because 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 file locks, and bugs that are discovered after the relevant stack frame is popped.

I. Introduction

Computer programmers are all-too-familiar with the task of debugging complex software through a combination of detective work, paperwork, and systematic search. Tracking down a bug usually starts with narrowing down the error to a specific component or even a file. The programmer¹ then sets to start from the final 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 source. Debuggers implement one or more bug triggers or trigger mechanisms to examine the state of the program at a given point in its execution. Armed with

¹In this paper, “programmer” refers to a person who is writing the application code. “User” refers to the person who is running the application, but is not involved in the programming task itself.

this information, the programmer then re-runs the program, steps it at or near the point in its execution history, observes the state at this point, and repeats the process.

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 can run for long periods of time; the art of debugging may persist their state, and they interact directly with hardware devices.

First, the act of debugging may persist the state of the operating system. The operating system is often a critical part of the system being debugged. 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 special privilege levels). In addition, the kernel of the system being debugged depends on some basic functionality in the debugged OS, such as reading and writing memory locations, setting and handling breakpoints, and communicating with the source debugger (e.g., through the serial line). Unfortunately, basic functionality may be implemented in the OS. In addition, some functionality may be provided by the hardware, such as memory protection across the OS to access hardware devices, and this functionality may not work on a real OS.

Finally, operating systems interact directly with hardware devices. Devices are sources of non-determinism that hinder cyclic debugging: they return data and generate events at unpredictable, irregular intervals. Devices may also fail due to transient errors, such as when a programmer during a debugging session.

“...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.

Debugging operating systems with time-traveling virtual machines

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

Abstract

Operating systems are difficult to debug with traditional cyclic debugging. They are non-deterministic; they can run for long periods of time before they interact directly with hardware devices. They are also non-transparent to the user of the system being debugged. 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 OS instance, and replay arbitrary segments of the past execution. We describe how to build a time-traveling debugger to enable a programmer to debug an OS in reverse, implementing commands such as reverse breakpoints, reverse watchpoints, and reverse single step. The space and time overheads needed to support time travel are measured. The results show that the overheads in time are low enough to support interactive debugging. The authors demonstrate the value of our time-traveling virtual machine by using it to understand a few serious 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 in memory management, and bugs in interrupt handling. The authors also show how to use time travel to trigger bugs that corrupt stack frames or are obscured after the infected stack frame is popped.

I. Introduction

Computer programmers are all-too-familiar with the task of debugging complex software through a combination of detective work, patterned and systematic search. Tracking down a bug usually starts with narrowing down the error to a specific component or even a file. The programmer¹ then sets to start from the final 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 source of an error. A debugger works by repeatedly jumping back to a previous instruction, and then jumping forward to the next instruction 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, steps it at an earlier point in its execution history, observes the state at this point, and then continues the process.

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

Traditional cyclic debugging is also problematic because

the interaction is effected by non-deterministic events such as

the interleaving of multiple threads, interrupts, user input, network I/O, and the permutations of states caused by the programmer who is debugging the system. This article describes how to use a time-traveling virtual machine to overcome many of the difficulties associated with the programming art of reverse engineering the state of an earlier point in time.

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

Third, the art of debugging may perturb the state of the operating system. The programmer is often a time-share user of the operating system. 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; instead the debugger uses special kernel-level interfaces to interact with the OS. The time-share kernel debugger depends on some basic functionality in the debugged OS, such as reading and writing memory locations, setting and handling breakpoints, and communicating with the source debugger (e.g., through the serial line). Unfortunately, basic functionality may be implemented differently in different OSes. In addition, some functionality may be provided by 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 events at unpredictable, changing intervals. Different devices also fail due to transient errors, such as if a program runs power 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

In addition, memory-mapped I/O instructions, and DMA memory loads. To avoid confusing the device, RIVM suppresses output to the device during replay.

The VMM must also be modified to support running real device drivers in the guest OS. Supporting MMU/T2T instructions is straightforward since they are privileged and safely trap to the VMM. After installing a trap from an MMU/T2T instruction, T2VM switches the processor and then reads the instruction in the device. After the instruction is executed, T2VM immediately places the switch back to the guest. Like MMU/T2T instructions, interrupt handling requires some modifications. T2VM already uses signals in place of hardware interrupts, so it forwards the interrupt using interrupt handlers, so it's forwarded to the guest using interrupt handlers.

To support memory mapping [19] and DMA, we implement the guest OS's memory mapping and DMA allocation routines to request access to the host's physical memory by issuing system calls to the host. For memory-mapped I/O, the guest OS asks the host to map the desired I/O region into the guest OS's address space. On MMU, the guest OS asks the host to allocate physical memory blocks at specific addresses. These allocations are managed by the guest in its own virtual address space. However, because the host must log all loads from the resulting virtual address range, the guest cannot have unchecklocked access to the newly allocated resources. As a result, T2VM uses page protection to trap all interactions with the protected virtual memory range. Upon receiving a trap, T2VM checks all guest pages in the block and sends that information with unchecklocked I/O space or DMA memory range. This provides sufficient opportunity to log and replay all interactions between the guest driver and the device.

The shortcoming of this approach is that the extra page and logging operations increase latency and wastes memory bandwidth and DMA memory. In practice, this increases the amount of time each bulk transfer is implemented using CPU requests. *copying* instructions, as bulk transfers cause only a single trap. We experienced no noticeable slowdown as a result of using this mechanism. For example, a guest OS's serial port driver can operate at full speed, and the guest OS sound card can play an MP3 audio clip and record audio in real time.

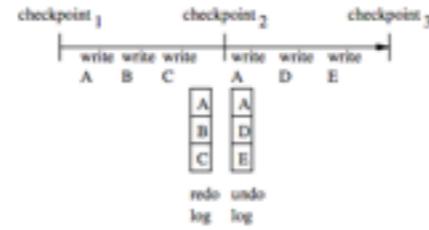
Allowing the guest device driver to initiate DMA transfers allows the guest OS to potentially corrupt host memory, since the device can access all of the host's physical memory [17]. The programmer who is won-

dered and you get more protection code, like a compiler and its runtime and a code that's printed on one end, and are read into the witness and you look for the violation, pointing right

to the code that you identify because that's what has been recorded by hardware. Indeed, the common knowledge under the assumption large buffer error is that your program contains

the code that's been recorded by hardware.

Figure 3: Checkpoints of the memory pages are represented as undo and redo logs. The figure shows the undo and redo logs that would result for checkpoints for the given sequence of writes to memory pages. The same technique is used to store the changes to the mappings of guest-to-host disk blocks.



3.3 Checkpointing for faster time travel

Logging and replaying a virtual machine from a single checkpoint at the beginning of the run is sufficient to resume the state at any point in the run from any other point in the run; however, logging and replay alone is not sufficient to resume this state quickly because the virtual machine must re-execute each instruction from the beginning to the desired point, and this process may take many steps. We propose time travel, which lets a user step through a virtual machine's execution without having to wait for the entire machine to run again [23].

T2VM takes periodic checkpoints while the virtual machine is running [23]. (ReVIR started very from a disk dumpfile of a powered-off virtual machine.)

The simplest way to checkpoint the virtual machine is to save a complete copy of the state of the virtual machine. This state is comprised of the CPU registers, the state of memory pages, the state of memory, the virtual disk, and the state in the VMM or host kernel. This offers the best recovery of the virtual machine. For UML, this free-hand state includes the address-space mappings for the guest host process and the guest kernel/host process, the state of open host file descriptors, and the registration of various signal handlers (including to the interrupt descriptor table and thread descriptor).

Having a complete copy of the virtual machine state is simple but inefficient. We use copy-on-write and checkpointing to reduce the space and time overhead of checkpointing for both memory pages and disk blocks. We use copy-on-write on memory pages to zero only those pages that have been modified since the last checkpoint.

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?

3 Time-traveling virtual machines

A few running virtual machines should have two main benefits. First, it should be easier to reconstruct the complete state of the virtual machine at any point in a trace, where a point is defined as the time from when the virtual machine was created until the last time it was modified. Second, it should be easier to analyze how many points a run and thus how many points were from the same instant that was measured during the original run without that point. The user describes how TIVM achieves these capabilities through a combination of logging, indexing, and checkpointing.

3.3 Logging and replaying a VM

The fundamental capability in TTVN is that the ability to run a guest in a given point in a way that matches the original user instruction for execution. Replay causes the virtual machine to transition through the same states it went through during the original guest, hence replaying the guest's execution. The guest can be suspended at any point in time. TTVN uses the built-in triggering mechanism to provide this capability [6]. This feature briefly summarizes how RAVI logs and replays the execution of a virtual machine.

A virtual machine can be replicated by starting

from a checklist, then reporting all sources of information (1, 2). See Table 1 for sources of information used during the checklist process.

are mapped to memory regions from the network, hardware, and memory check and the issuing of virtual interrupt. The VMM thus reads memory and hardware input by tagging the cells that map these devices during the initial set-up and re-implements the same data during the update. Likewise, we configure the CPU to cause reads of memory and memory access to map to the VMM, where they can be checked.

3.3 Host device drivers in the guest OS

In general, VMEbus can be limited by virtual devices. VMEbus virtual devices have been in use for many years. VMEbus Virtual Device exports as well as VMEbus Virtual Device imports are supported. VMEbus Virtual Device imports (also called VMEbus Virtual Device drivers) are typically implemented as host drivers. VMEbus Virtual Device drivers are typically limited to one or two virtual devices. This is because each virtual device needs memory space reserved for itself. However, when using virtual machines in a QEMU operating system, the limited number of virtual devices per processor is overcome by using additional memory for real devices. Processor programs can continue addressing the architecture-independent portion of the guest OS. There are three ways to achieve this: 1) adding a virtual device to a guest OS and letting host devices access it via a port. 2) With host memory, a virtual device driver can be included in the guest OS without being modified or recompiled.

The first way to run a user-defined driver in the guest OS is to use the VMFS to provide a software emulator for the device. The device driver loader runs the command in the guest OS to load the driver code. The VMFS provides the RAVP instructions, memory mapping, DMA, MMIO commands, and so on. The VMFS also provides the interrupt controller and timer for the software driver emulation. With this method, RAVP can log and replace the driver code in the same way as it does for the host OS. If one uses the VMFS to run the driver code, the VMFS must support the guest operating system and handle the configuration system described in section 3.1.3. MyRAV will guide the emulator and the driver code through the user-interaction message to make the driver code as well as the VMFS do the necessary logging. While this the method is simple and effective, it is not efficient. The only work it does is to have an accurate software emulator to run the device whose driver runs within a virtual machine.

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.

3. Time-traveling virtual machines

A time-traveling virtual machine should have two capabilities. First, it should be able to record the execution of the virtual machine at any point in time so that when a run is defined in 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 thus that point replay the same instructions over that run starting from the same state of the virtual machine that was executing during the original run from that point. The authors describe how UVMV achieves these capabilities through a combination of logging, replay, and checkpointing.

3.1 Logging and replaying a VM

The fundamental capability in UVMV is the ability to log a run from a given point in a way that records 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 after one to many points in the run. The current state of the virtual machine is recorded at every point in the run. This enables ReViN to replay the scheduled data during the original run. ReViN also provides the ability to log and replay the real-time clock setting on the VM, where they can be logged or reprogrammed.

To replay a virtual memory, ReViN logs the instructions in the run at which it was defined and on delivery the memory at the same location during replay. This pause is achieved by recording the address of branches over the start of the run and the address of the instruction after the instruction [15]. ReViN 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 interrupt timer logic to stop at the instruction after the last branch. ReViN uses the scheduling order of could threads past operating systems and applications, as long as the VM is using the abstraction of a uniprocessor virtual machine [22]. Researchers are investigating ways to support replay on multiprocessors [36].

3.2 Host device drivers in the guest OS

In general, VMs require a limited set of virtual devices. Some VMs support virtual devices that exist in hardware (e.g., VMware supports an emulated AMD-Lane Ethernet card); others (like UML) support virtual devices that have no hardware equivalent. Supporting a limited set of virtual devices in the guest OS is a limitation of current virtual machine monitors. The reason is that there are three needed device drivers for typical host devices [28]. However, when using virtual machines in debug operating systems, the limited set of virtual devices is a significant problem from using and developing drivers for real hardware, preventing the developer from writing 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 mapped to the guest OS without being modified.

The first way is to use a real device driver in the guest OS for the VM to provide a software emulator for that device. The device driver issues the normal set of I/O instructions (read/write instructions, memory mapped I/O, DMA commands, and so on). The VM then maps the device driver to the guest OS and forwards requests to the software device emulator. With this strategy, ReViN can log and replay device driver code in the same way it logs and replays the rest of the guest OS. It can reuse the VM's software device emulator above ReViN's logging layer, which makes the development of the device driver much easier [31]. ReViN will provide the emulator and device driver code through the same instruction sequence during replay as they occurred during logging. While this first strategy fits in well with the existing ReViN system, it only works if one has an accurate software emulator for the device driver.

We modified UML to provide a second way to support device drivers in the guest OS, which works even when no software emulator exists for the device of interest. With this strategy, the VM logs and forwards the programmed I/O requests to the device driver located above the guest device driver to the actual hardware. The programmer specifies which device UML can access, and the VM will interface the proper I/O port space and memory access for the device.

This second strategy requires extensions to modify ReViN to log and forward the requests of the device driver to the device driver itself. The device driver is located above the ReViN logging layer; the second strategy requires write-driver actions to the actual hardware device. Because this device may not be deterministic, ReViN must log any information sent from the device to the driver specifically. ReViN must log and replay the data received

“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.

3. Time-traveling virtual machines

A time-traveling virtual machine should have two capabilities. First, it should be able to record the execution of the virtual machine at any point in time so that when 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 thus that point replay the same instructions over that was executed during the original run from that point. The authors describe how TTVMS achieves these capabilities through a combination of logging, replay, and checkpointing.

3.1 Logging and replaying a VM

The fundamental capability in TTVMS is the ability to log a run from a point prior to a run that records 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 after one to many runs of the same state of the virtual machine at any point in the run. The authors of ReViMS logging mechanism is provide this capability [9]. This section briefly summarizes how ReViMS logs and replays the execution of a virtual machine.

A virtual machine can be replays its starting from a checkpoint, thus recording all source of non-determinism. In this case, the VM will be able to determine are external input from the network, key board, universal-time clock and the timing of virtual interrupt. The VMMS replays network and keyboard input by logging the calls that read these devices during the original run and inserting the same data during the replay. In addition, the VMMS replays the universal-time clock and the real-time clock stored in the VMMS, where they can be logged or reproduced.

To replay a virtual interrupt, ReViMS logs the instruction in the run at which it was delivered and on delivers the interrupt at the same location during replay. This pause is achieved by identifying the location of branches over the start of the run and the address of the instruction of the instruction [10]. ReViMS uses a performance counter on the host (Precision 4 CPU) to count the number of branches during logging, and it uses the same performance counter and interrupt timer register to stop at the instruction of interest during replay. ReViMS uses the scheduling order of each thread past operating systems and applications, as long as the VMMS respects the abstraction of a uniprocessor virtual machine [22]. Researchers are investigating ways to support replay on multiprocessors [36].

3.2 Host device-drivers in the guest OS

In general, VMMS exports 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 is a good idea because it limits the number of virtual devices that the guest OS needs to handle simultaneously. However, it leaves gaps. Off-the-shelf device drivers for typical host devices [28]. However, when using virtual machines in debug operating systems, the limited set of virtual devices is a major problem from using and developing drivers for real devices, preventing the developer from debugging 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 integrated in the guest OS without being modified.

The first way is to use a real device driver in the guest OS for the VMMS to provide a software emulator for that device. The device driver issues the normal set of I/O instructions (XEN/PCI instructions, memory mapped I/O, etc.) commands, and the VMMS translates those commands and forwards them to the software device emulator. With this strategy, ReViMS can log and replay the rest of the guest OS. If one uses the VMMS's software device emulator above ReViMS's log and replay mechanism, the device driver must be modified to work with the existing ReViMS system. The problem is that the device driver must be modified to work with the existing ReViMS system, it only works if one has an accurate software emulator for the device driver.

We modified UML to provide a second way to support device drivers in the guest OS, which works even when no software emulator exists for the device of interest. With this strategy, the VMMS exports and forwards the programmed I/O requests from the device driver to the real-world device driver to the actual hardware. The program specifies which device UML can access, and the VMMS enforces the proper physical space and memory access for the device.

This second strategy requires extensions to modify ReViMS to log and replay the data of the device driver. The first strategy allows the device driver to be located above the ReViMS logging layer, the second strategy allows the device driver to be located below the ReViMS logging layer. Because this device may not be deterministic, ReViMS must log many information sent from the device to the driver specifically. ReViMS must log and replay the data received

“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?

6). User-Wide Linux handles interrupts and preemptions with asynchronous signals, and prior research has shown we are not able to capture such events. In addition, interrupt handlers are not yet able to support the logging of other events in isolation [26, 1, 23, 4], and the overhead for each event is significant [26, 1, 23, 4], and the overhead for each event is significant [26, 1, 23, 4], and the overhead for each event is significant [26, 1, 23, 4]. Finally, some systems work at the language level [27], and this process takes them as the language level [27].

Researchers have worked to capture non-deterministic programs through various approaches. The events of different threads can be explored at different levels, increasing logging accuracy to shared objects [38], keeping an eye on the state of memory objects or program code or environment [22], or logging physical memory access in hardware [2]. Other researchers have worked to update the amount of data logged [21].

Virtual machine logging has been used for non-debugging purposes, such as monitoring system logs to capture the state of a backup machine to provide fault tolerance [25]. Relying on virtual-machine replay to enable detailed forensic analysis [35]. One work approach is to use replay to achieve a more rapid fix, which is a more direct approach than simulations. TTVVM also requires additional features over prior virtual machine replay systems. TTVVM requires the ability to run, log, and replay end-device drivers in the guest OS, allowing prior system mechanics to run on real hardware instead of virtual machines in the guest OS. In addition, TTVVM can travel quickly forward and backward in time through the use of checkpoints and undo and redo logs, whereas ReVive requires only a single snapshot of a virtual machine and machine and replay driver did not need to remember time travel at all as only replicated copies within one space.

Another approach for providing time travel is to use a computer-machine simulator, such as Nitro [36]. Nitro supports deterministic replay for memory operations and supports both forward and backward time travel. However, Nitro is drastically slower than TTVVM, and the replay debugging time is impractical. On a 700MHz UltraSparc III, Nitro consumes 2+ million CPU instructions to replay one second of recorded time down to 1ms [36], whereas virtual machines typically incur a slowdown of less than 2x.

8 Conclusions and Future Work

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

TTVVM added reasonable overhead in the cost of running the debugger. The debugger overhead was 1.5 times the time OS spent in checkpoints, added 3-12% in replaying time and 2-6% Kbytes in log space. Taking checkpoints every minute added less than 9% time overhead and 1.1 Million space overhead. Taking checkpoints every second added a portion of time overhead to 10-15% and resulted in replaying a portion of time overhead to completion in about 12 seconds.

We used TTVVM and our new remote debugging commands to fix four OS bugs that were difficult to find with traditional debugging. We found the use of the log file greatly enhanced our ability to identify and fix the bugs in use. Remote debugging proved especially helpful in finding bugs that were fragile due to non-deterministic bugs in device drivers. Bugs that required long runs to trigger them were often triggered by the log file and were easier to fix once the relevant stack trace was printed.

Possible future work includes exploring more efficient debugging operators that are modified by user input and system feedback. For example, one could automatically turn off the effects of a program modification change by taking the checkpoint and comparing the results after the change with the results of the original run.

9 Acknowledgements

Our shepherd, Steve O'Farrell, and the anonymous reviewers provided feedback that helped improve this paper. This work was supported in part by AFOSR grant FA8650-04-2-0004, National Science Foundation grants CCR-0096229 and CCR-0230003, and by Intel Corporation. Instead King was supported in part by a National Defense Science and Engineering Graduate Fellowship.

References

- [1] H. Agius, R. A. Sekhon, and T. W. Nalebuff. An Efficient Algorithm for Program Rewriting. *ACM SIGART Letters*, 4(2), May 1990.
- [2] D. T. Baur and E. C. Goldberg. Hardware Assistant for Help of Maintenance Programs. In *Proceedings of the International Conference on Parallel and Distributed Debugging*, 1991.
- [3] P. Bellone, R. Dragozzi, R. Russo, R. Rossi, P. Saccoccia, and G. Vassalli. A New and Efficient Algorithm for Program Rewriting. In *Proceedings of the 1990 International Conference on Parallel Processing*, 1990.
- [4] B. Beeton. Efficient algorithms for bidirectional string maps. In *Proceedings of the 2000 Conference on Programming Languages Design and Implementation (PLDI)*, pages 389–410, June 2000.

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.

6) User-Wide Linux handles interrupts and preemptions with asynchronous signals, and prior research demonstrates we are not able to capture such events. In addition, interrupt handlers are not yet able to support the logging of other events in isolation [26, 1, 23, 4], and this limits the ability to log each data item every minute without impacting system performance such as CPU. Finally, some systems work at the language level [27], and this places them at risk when running on a system in a different language or with different libraries.

Researchers have worked to explore non-deterministic programs through various approaches. The events of different threads can be explored at different levels, including logging across shared objects [28], looking at the execution of memory operations [29], or instrumentation [30], or logging physical memory access in hardware [31]. Other researchers have looked to update the amount of data logged [32].

Virtual machine logging has been used for non-debugging purposes, such as monitoring system activity to determine the state of a backup machine to provide fault tolerance [33], R/W file-level virtual-machine replay to enable detailed forensic analysis [34]. One work approach is to replay to determine a bug report, which is a reverse approach to what TTVMS does.

TTVMS also requires additional features over prior virtual machine engine versions. TTVMS requires the ability to run, log, and replay end-device drivers in the guest OS, although prior virtual machines did not support running drivers in the guest OS.

In addition, TTVMS can travel quickly forward and backward in time through the use of checkpoints and undo-redo logs, whereas ReVive requires only a single snapshot of a virtual machine and requires a host driver that are used to maintain time travel at all as only registered guests within one guest.

Another approach for providing time travel is to use a complete machine emulator, such as QEMU [35], which supports discrete logging for memory operations and supports both forward and backward in time. However, QEMU is drastically slower than TTVMS, and this makes debugging long runs impractical. On a 700MHz UltraSparc III, QEMU consumes 2x million CPU instructions per second (measured times slower than native [36]), whereas virtual machines typically incur a slowdown of less than 2x.

8 Conclusions and future work

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

TTVMS added reasonable overhead in the context of memory access. The logging of memory data overhead is less than 10% for memory operations with 1-100ms running time and 2-85 KB/sec in log space. Taking checkpoints every minute added less than 9% time overhead and 1-5 MB/sec space overhead. Taking checkpoints are only 1-2% overhead and enables a portion of our applications to complete in about 12 seconds.

We used TTVMS and our new reverse debugging commands to fix four QEMU bugs that were difficult to find with traditional debugging. We found that the use of logging greatly enhanced our ability to identify and fix these bugs. Reverse debugging proved especially helpful in finding bugs that were fragile due to race conditions, bugs in device drivers, bugs that required long runs to trigger, and bugs that required specific memory access patterns that the reverse-debug trace was prepared.

Possible future work includes exploring more advanced debugging operations that are enabled by time travel and checkpoints. For example, one could compare the effects of a programmatic environment change by taking the checkpoint and comparing the results after the change with the results of the original.

9 Acknowledgements

Our shepherd, Steve O'Farrell, and the anonymous reviewers provided feedback that helped improve this paper. This work was supported in part by AFRL grant FA8650-04-2-0014, National Science Foundation grants CCR-0096229 and CCR-0230061, and by Intel Corporation. Instead King was supported in part by a National Defense Science and Engineering Graduate Fellowship.

References

- [1] H. Agius, R. A. Sekhon, and T. W. Nalebuff. An Efficient Algorithm for Time-Travel Debugging. *ACM SIGART Newsletter*, 4(3), May 1996.
- [2] D. T. Bouc and E. C. Grindstaff. Hardware Assisted Recovery of Performance Problems. *In Proceedings of the International Conference on Parallel and Distributed Processing*, 1993.
- [3] P. Bellone, R. Dragozzi, R. Russo, R. Rossi, P. Saccoccia, R. Negrianni, J. Piro, and A. Martini. Not and the Art of Visualizing. *In Proceedings of the 10th International Conference on Parallel and Distributed Processing*, 1998.
- [4] B. Beeton. Efficient algorithms for bidirectional tracing. *In Proceedings of the 2000 Conference on Programming Languages Design and Implementation (PLDI)*, pages 389–410, June 2000.

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

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

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

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.

-
- ➊ GDB: linked list storage of every memory op
 - + Easy to build and reason about
 - Both recording and replaying is very slow
 - ➋ 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
 - ➌ King (et al): Only records external, non-deterministic events
 - + Reduces log size and improves performance
 - synchronizing external events becomes complicated

Improving performance with JIT compilation



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 instrumentation 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

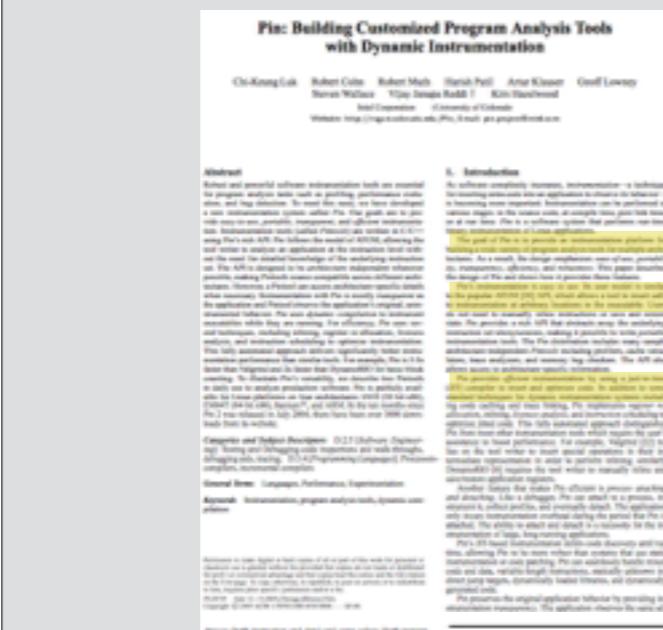
PLDI '05



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



“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.

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

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.



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

The screenshot shows the title page of a research paper. The title is "Luk et al. Pin: Building Customized Program Analysis Tools w/ Dynamic Instrumentation". Below the title is the conference name "PLDI '05". The main content area contains the abstract, which discusses the Pin tool for dynamic instrumentation. It includes code snippets in C and assembly language, showing how memory writes are instrumented. A figure caption "Figure 1. A Pin tool for tracing memory writes." is present, along with a note about the original code being modified for the Pin tool.

Figure 1. A Pin tool for tracing memory writes.

Abstract: Pin is a dynamic instrumentation framework for x86 programs. It provides a C API for instrumenting assembly code, and a C API for instrumenting C code. Pin can be used to build customized program analysis tools, such as memory leak detectors, performance monitors, and security scanners. Pin is designed to be efficient and flexible, allowing users to easily add new instrumentation points or modify existing ones. Pin is also designed to be easy to use, with a simple API and a clear documentation.

Figure 1 shows the source code for a Pin tool that traces memory writes. The code uses the Pin API to instrument memory write operations. It includes a main function that initializes the Pin API, opens a trace file, adds an instrumentation function, and starts the program. The instrumentation function is called for every instruction, and it checks if the instruction is a memory write. If it is, it records the address and size of the write. The code then prints the trace to the console.

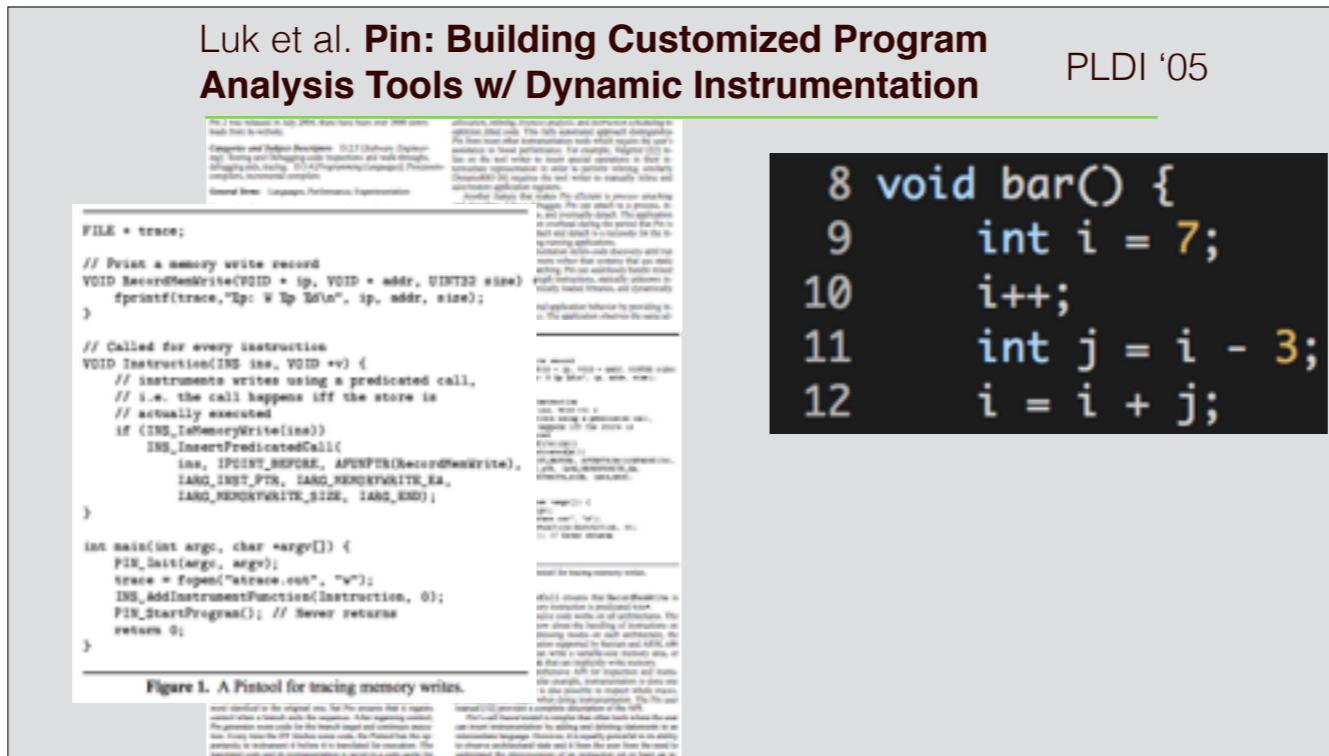
Figure 1 also shows the assembly code generated by Pin. The assembly code includes calls to the Pin API to record memory writes and to print them to the console. The assembly code is annotated with comments explaining the purpose of each instruction.

Figure 1. A Pin tool for tracing memory writes.

Figure 1 shows the source code for a Pin tool that traces memory writes. The code uses the Pin API to instrument memory write operations. It includes a main function that initializes the Pin API, opens a trace file, adds an instrumentation function, and starts the program. The instrumentation function is called for every instruction, and it checks if the instruction is a memory write. If it is, it records the address and size of the write. The code then prints the trace to the console.

Figure 1 also shows the assembly code generated by Pin. The assembly code includes calls to the Pin API to record memory writes and to print them to the console. The assembly code is annotated with comments explaining the purpose of each instruction.

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.



```

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.

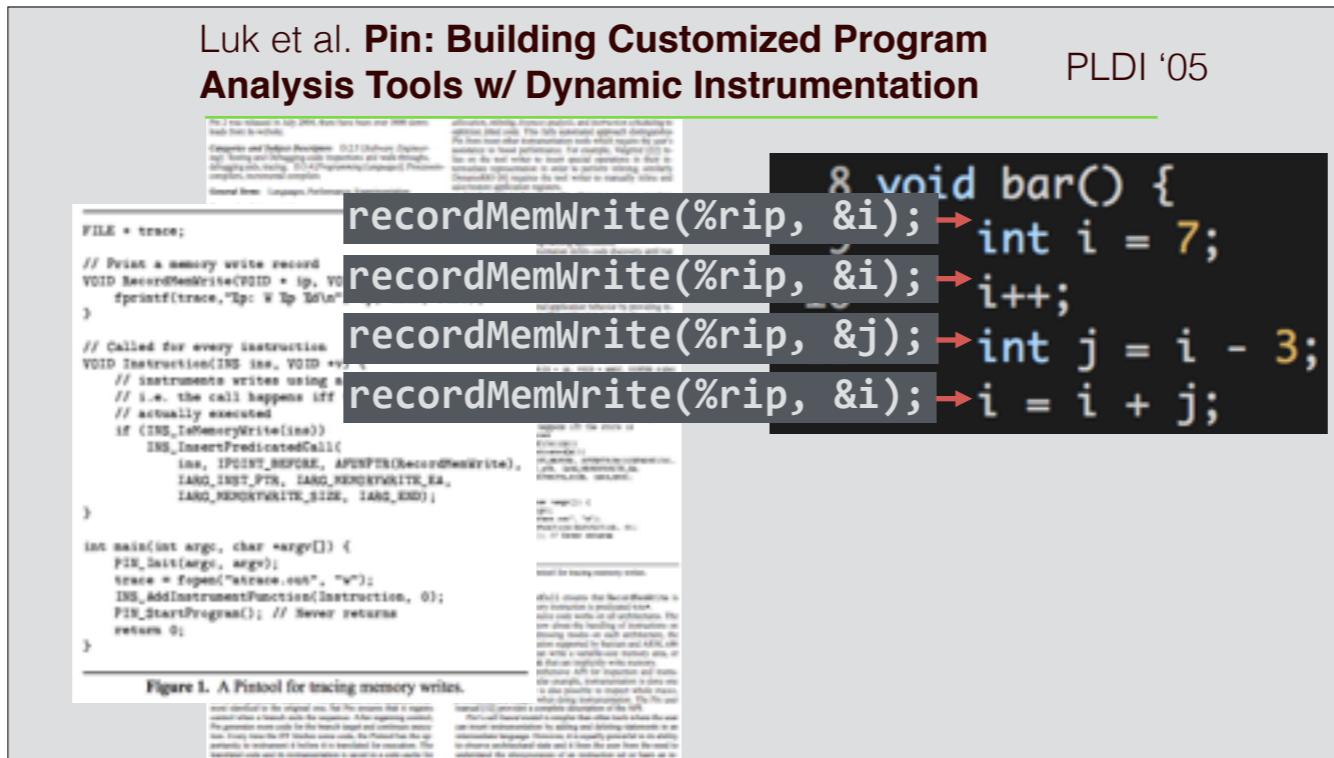
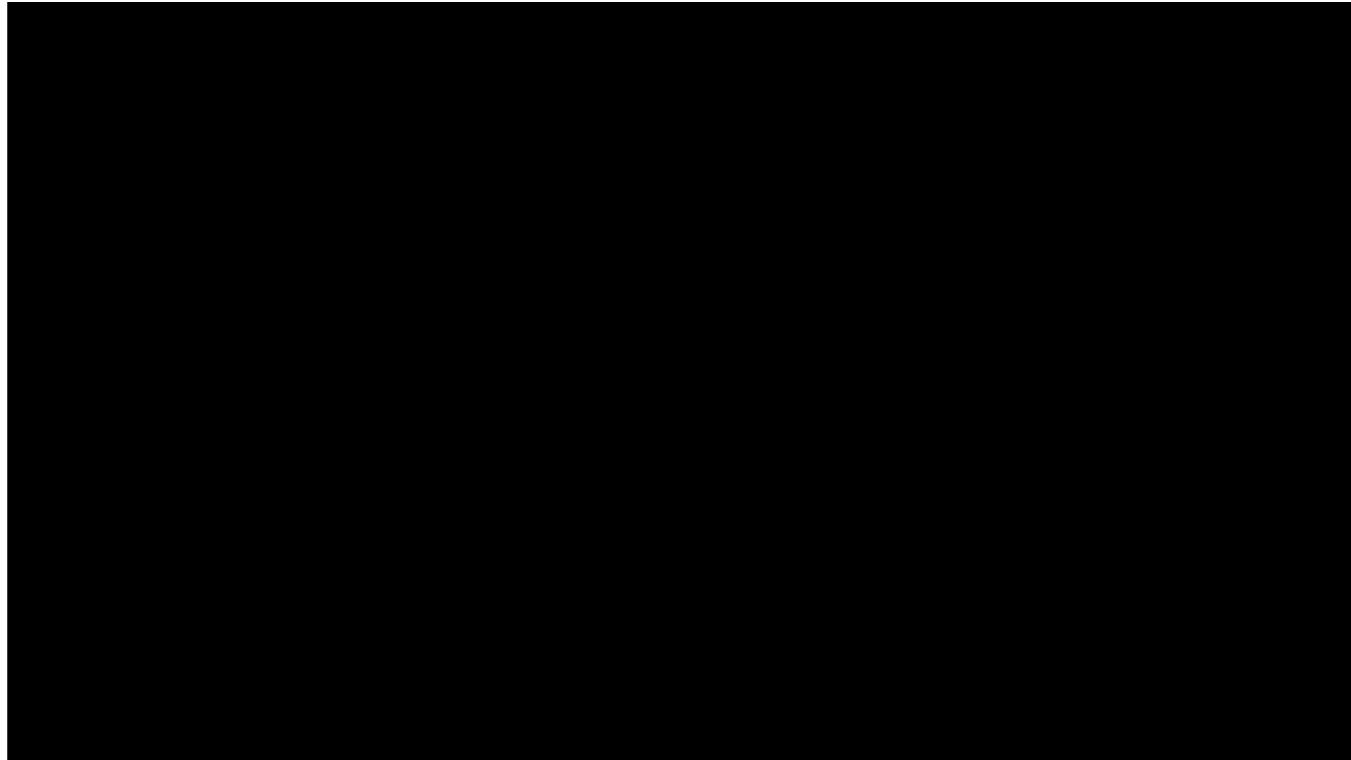


Figure 1. A Pin tool for tracing memory writes.

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.



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)
```

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.

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

```
TRACE_AddInstrumentFunction()
```

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

```
RTN_AddInstrumentFunction()
```

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

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>

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

IMG_AddInstrumentFunction()
push  %rbp
mov   %rsp,%rbp
sub   $0,%rbp
mov   $0,sub(%rbp)
movq  $0,sub(%rbp)
movq  $0,sub(%rbp)
add   $0,sub(%rbp)
mov   $0,sub(%rbp)
movq  $0,sub(%rbp)
movq  $0,sub(%rbp)
callq 0x200b10(%rip),%rax
mov   $0x2a,(%rax)
pop   %rbp
popq  $0,sub(%rbp)
callq 0x200b10(%rip),%rax
leaved
retq

```

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.

```

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

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

static void
instrument_expr (gimple_stmt_iterator gsi, tree expr, int is_write)
{
  enum tree_code tcode;
  unsigned fid_off, fid_size;
  tree base, rhs;
  gimple_stmt;
  gimple_seq gs;
  location_t loc;

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

  tcode = TREE_CODE (expr);

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

  if (tcode == COMPONENT_REF) {
    tree field = TREE_OPERAND (expr, 1);
    if (TREE_CODE (field) == FIELD_DECL) {
      fid_off = TREE_INT_CST_LOW (DECL_FIELD_BIT_OFFSET (field));
      fid_size = TREE_INT_CST_LOW (DECL_SIZE (field));
      if (((fid_off % BITS_PER_UNIT) != 0)
          || ((fid_size % BITS_PER_UNIT) != 0)) {
        /* As of now it crashes compilation.
         * TODO: handle bit-fields as if touching the whole field.
         */
        return;
      }
    }
    /* TODO: handle other cases
     * (FIELD_DECL, MEM_REF, ARRAY_RANGE_REF, TARGET_MEM_REF, ADDR_EXPR).
     */
    if (tcode != ARRAY_REF
        && tcode != VAR_DECL && tcode != COMPONENT_REF
        && tcode != INDIRECT_REF && tcode != MEM_REF)
      return;

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

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

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

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

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

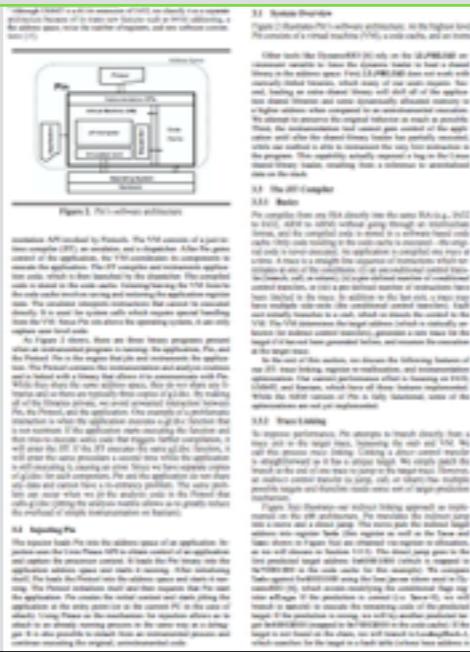
thread-sanitizer/blob/master/gcc/tree-tsan.c

```

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.



“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

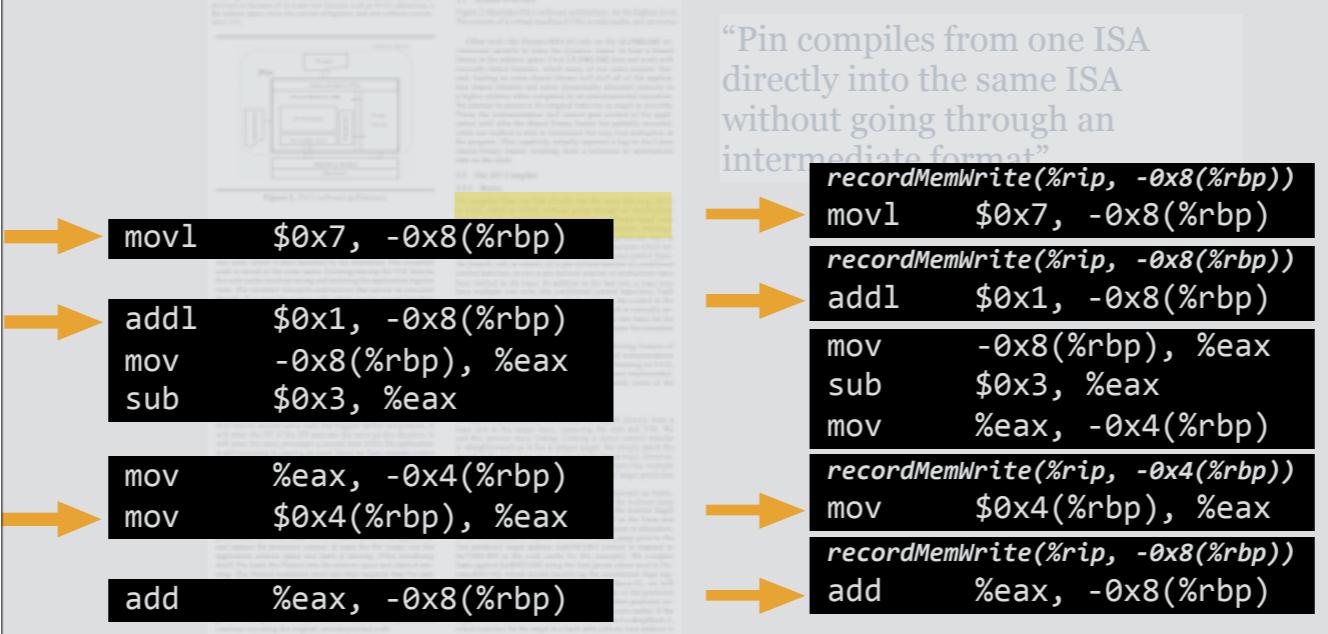
“Pin compiles from one ISA directly into the same ISA without going through an intermediate format”

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

“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.



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

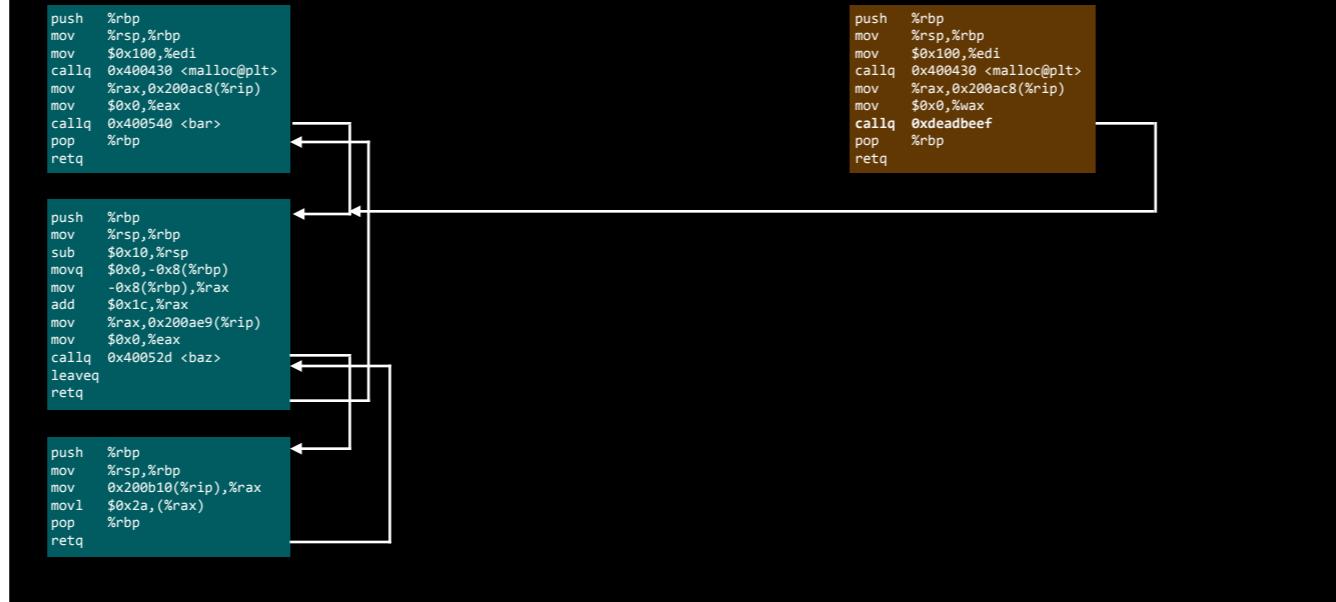
```
push %rbp  
mov %rsp,%rbp  
mov $0x100,%edi  
callq 0x400430 <malloc@plt>  
mov %rax,0x200ac8(%rip)  
mov $0x0,%eax  
callq 0x400540 <bar>  
pop %rbp  
retq
```

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

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

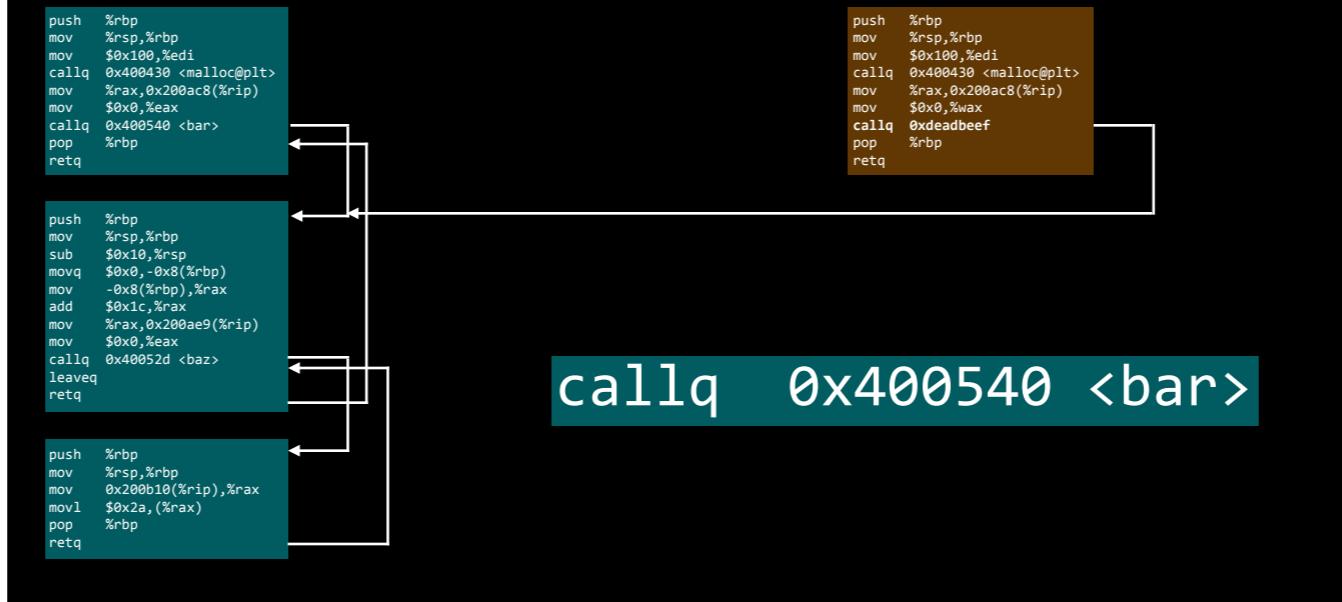
This rewriting happens lazily when we first begin to execute a block of code. So let's say it's the first time we're entering the top block -

jump rewriting



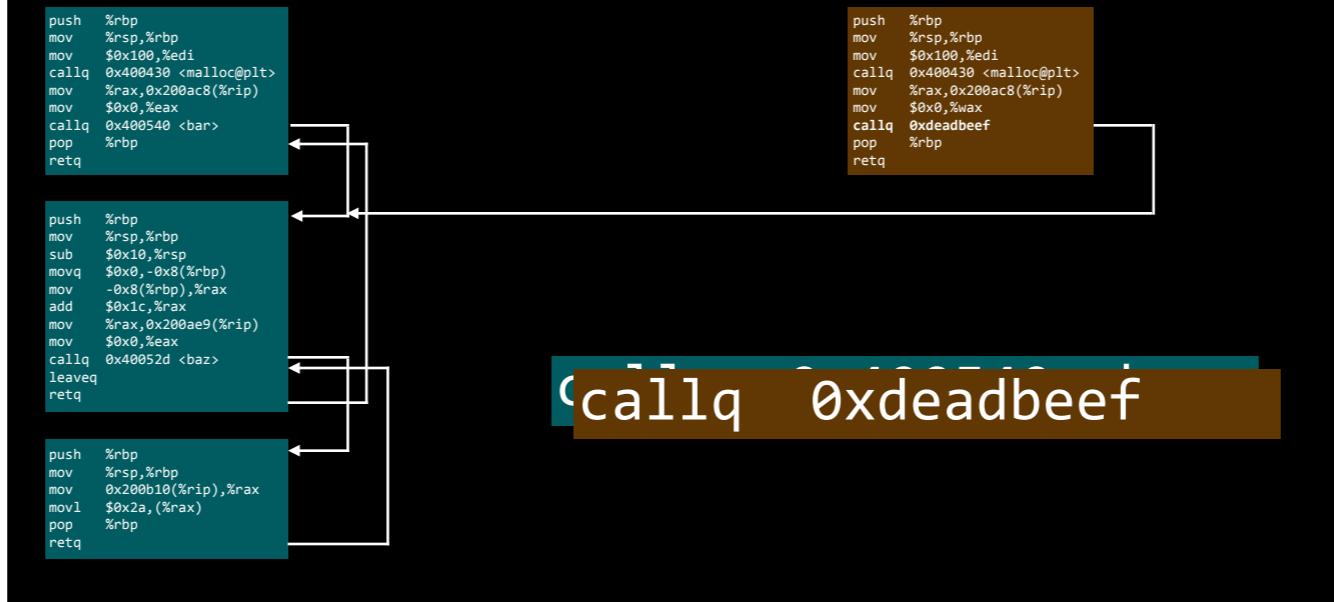
We copy the block and instrument it, but if we don't patch that control flow change, we could find ourselves jumping back into uninstrumented code! So what we do we <click> take the target address and <click> have it call back into Pin, which would in turn

jump rewriting



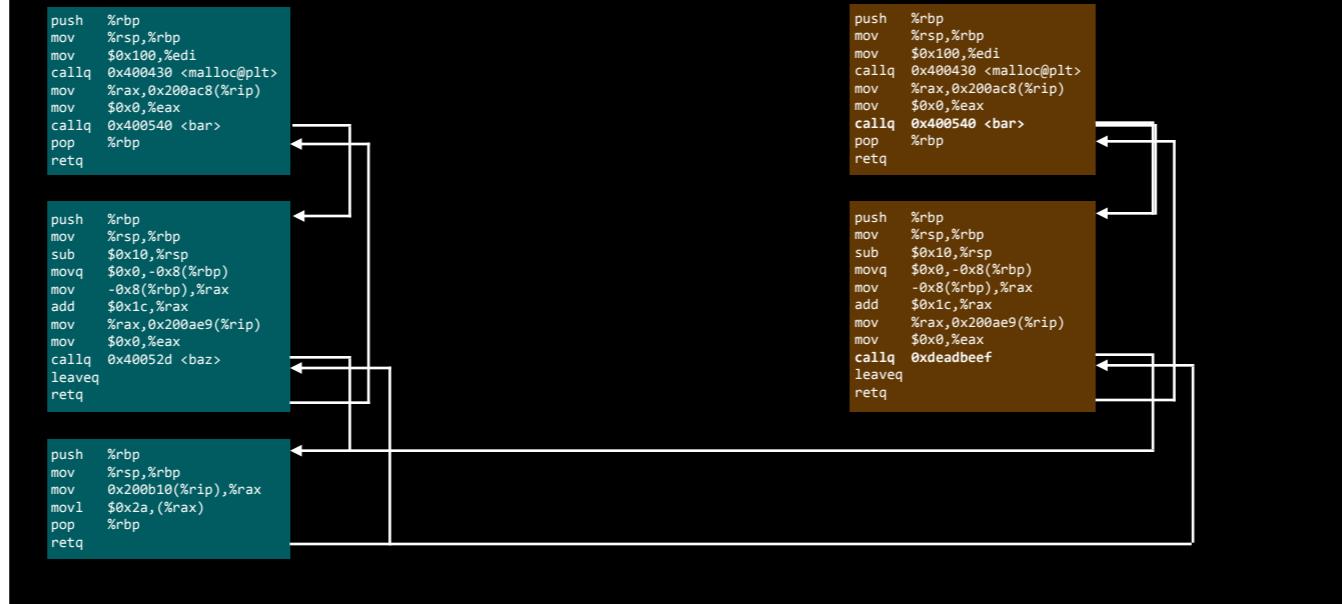
We copy the block and instrument it, but if we don't patch that control flow change, we could find ourselves jumping back into uninstrumented code! So what we do we <click> take the target address and <click> have it call back into Pin, which would in turn

jump rewriting



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
nicholas.nethercote@nicta.com.au

Julian Seward
Open Source Lab, Cambridge, UK
jseward@open-source.org

Abstract

Dynamic binary instrumentation (DBI) frameworks make it easy to build dynamic binary analysis (DBA) tools such as profilers and profilers. Much of the focus on DBI frameworks has been on performance. Little attention has been paid to their capabilities. This paper describes Valgrind, a DBI framework.

In this paper we describe Valgrind, a DBI framework designed for building heavyweight DBA tools. We focus on its unique cap-abilities, which include: (1) support for multiple languages and efficient multi-threaded DBA modules, which requires a tool to shadow every register and memory value with another value that matches the original access; (2) support for pointer analysis that tracks that shadowed pointer from one code location to the next; (3) support for tracking pointer aliasing between the code of three threads; (4) support for both full and light-weight instrumentation; (5) support for both shared and private memory; and (6) support for both DBI frameworks such as PIN and DynamoRIO.

Categories and Subject Descriptors: D.2.2 [Software Engineering]: Testing and Debugging—debugging aids, monitors; D.4.2 [Programming Languages]: Processors—instrumentation compilers.

General Terms: Design, Performance, Experimentation.

Keywords: Valgrind, instrument, dynamic binary instrumentation, dynamic binary analysis, shadow values

1. Introduction

Valgrind is a dynamic binary instrumentation (DBI) framework that is used to build dynamic binary analysis (DBA) tools such as profilers and debuggers. This paper describes how it works, and how it differs from other frameworks.

1.1 Previous Binary Analysis and Instrumentation

Many environments are program analysis tools, such as static checkers and provers, to improve the quality of their software. However, they do not instrument programs at run-time. Instead, they analyze programs at run-time at the level of machine code.

Other tools are often implemented using dynamic binary instrumentation, which instruments programs in run-time to inspect the state of the client programs at run-time. This is common for tools

as the debugger (such as GDB) or the linker (such as ld). Most of the focus on DBI frameworks has been on performance. Little attention has been paid to their capabilities. This paper describes Valgrind, a DBI framework.

The performance of DBI frameworks has been studied closely [3, 2, 4]. Low overhead is often the primary goal of DBI frameworks, but there is little work that studies the cost of the features 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 explored.

1.2 Shadow Value Tools: Heavyweight DBA

One main group of DBI tools are those that use shadow values. These tools store many in-memory values, many register and memory values with another value that says something about it. We call these shadow values. Consider the following monitoring task: monitor every pointer in memory to see if it is being modified. This means that shadow values can be used in a wide variety of ways, and (1) are powerful and interesting.

For example, shadow values can track where the values are modified (i.e., annotated), or derived from undefined values, and can thus detect dangerous use of undefined values. It is used by the Valgrind [1] and Pin [2] profilers, and is probably the most well-known DBI tool.

Valgrind [1] tracks which four values are tested (i.e., those in memory, or memory from the stack, or memory from the heap, or memory from the heap) and which values. However, PIN and DynamoRIO [2] are similar tools.

Valgrind [1] and Pin [2] have monitoring, and LTT [3] tools for which they are used. LTT is a performance monitor, and monitors how much information about recent memory is recorded by profile counters.

Memory [4] tracks each value's type (determined from static analysis) and its value, and can then use this to predict subsequent operations for a value of that type.

DynoCop [5] maintains detailed types of local variables, and can then use this to predict what values are in memory.

Address [6] tracks which word values are array pointers, and tracks this to detect buffer overflows.

Address [6] and Address [7] combine, produce a resolution of a program's static components, from the graph one can use all of the program's components to reach the each value's creation.

In these examples, the tools are a monitor, a type predictor,

and a type checker. In Valgrind [1], a monitor, a type predictor, and a type checker.

Pin [2] is a monitor, checker, and writer to Monitors. However, Pin [2] is not a monitor, checker, and writer to Monitors.

Valgrind [1] is a monitor, checker, and writer to Monitors.

Address [6] is a monitor, checker, and writer to Monitors.

Address [7] is a monitor, checker, and writer to Monitors.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

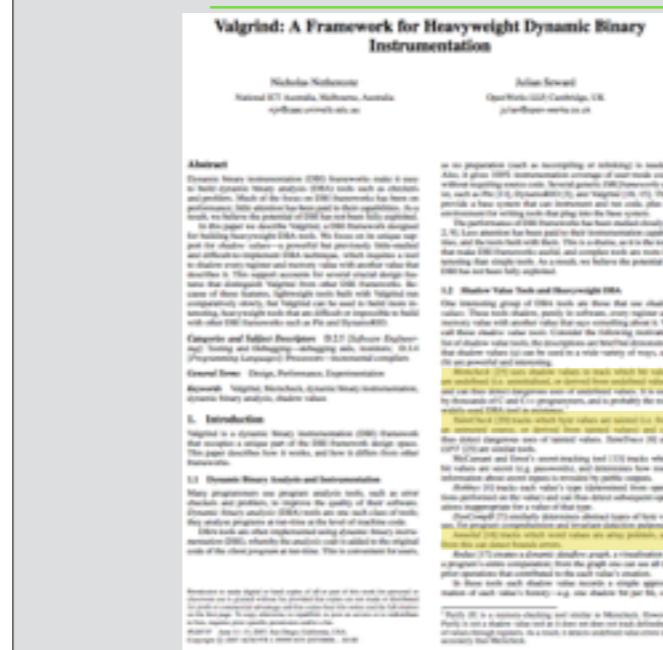
Address [6] and Address [7] are combined to get the final moni-

tor.

Address [6] and Address [7] are combined to get the final moni-

tor.

</



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

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

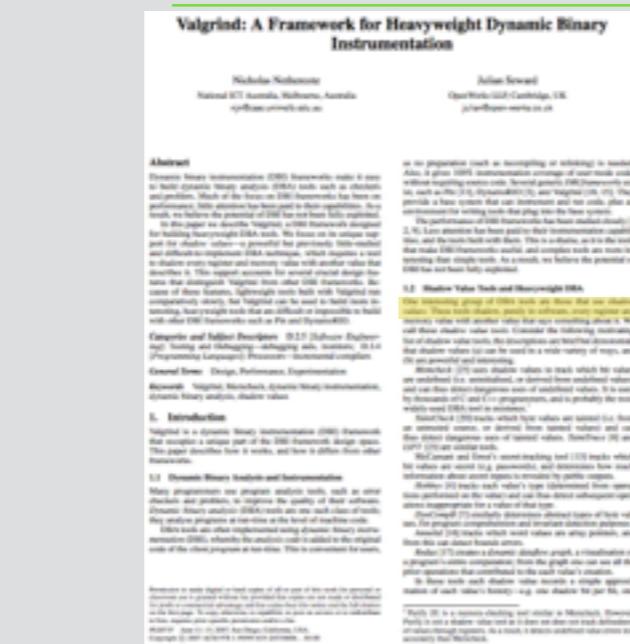
“which word values are array pointers, and from this can detect bounds errors.”

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

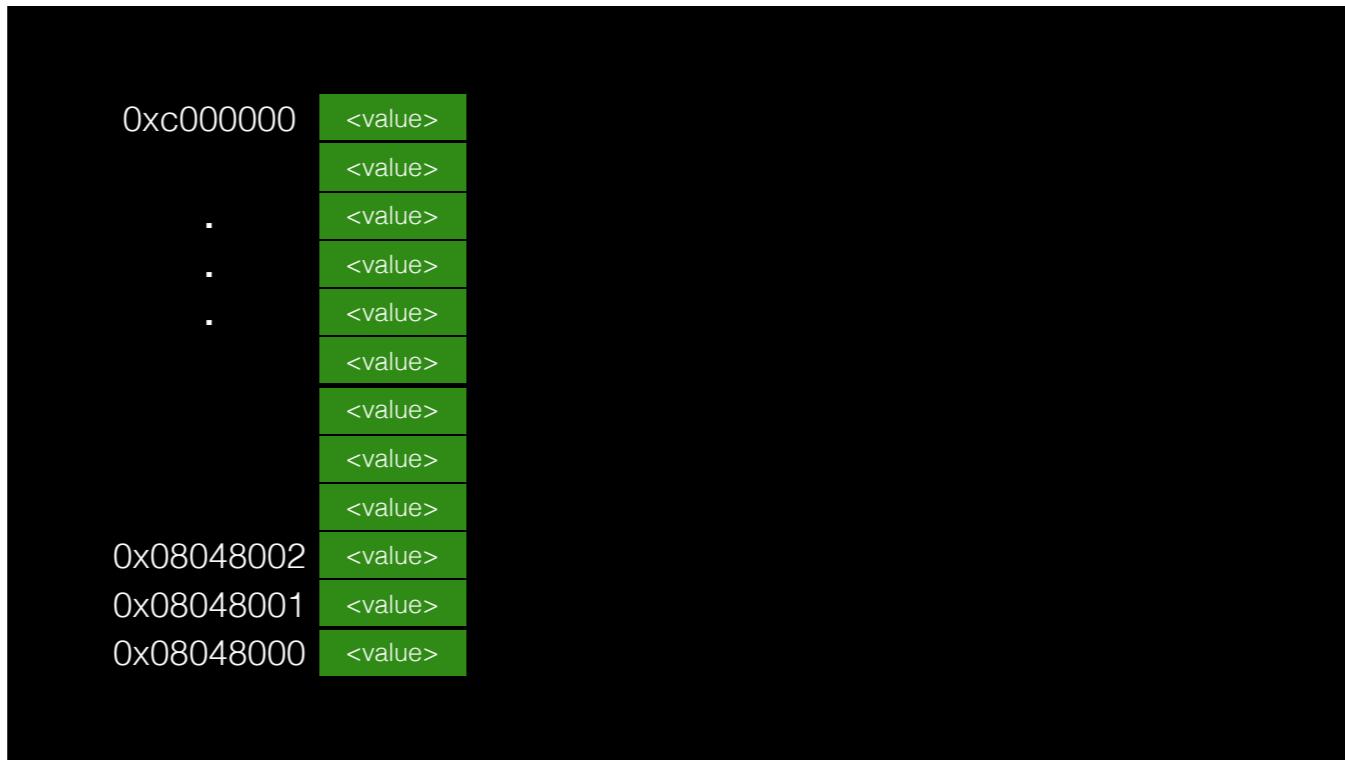
PLDI '07

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



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.



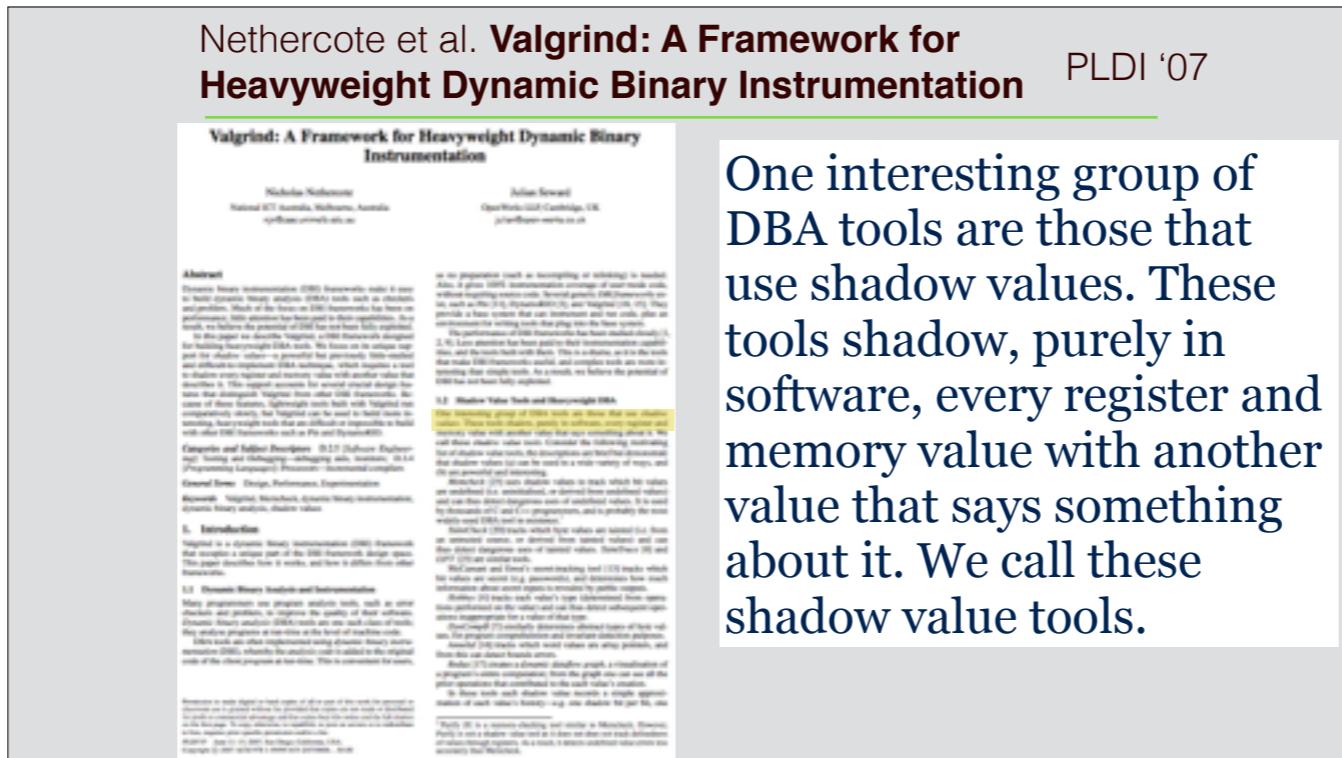
So if ordinarily memory can be thought of as a big array that spans the addressability of the entire process,



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



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.

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.

0xc000000



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

memcheck/mc_main.c

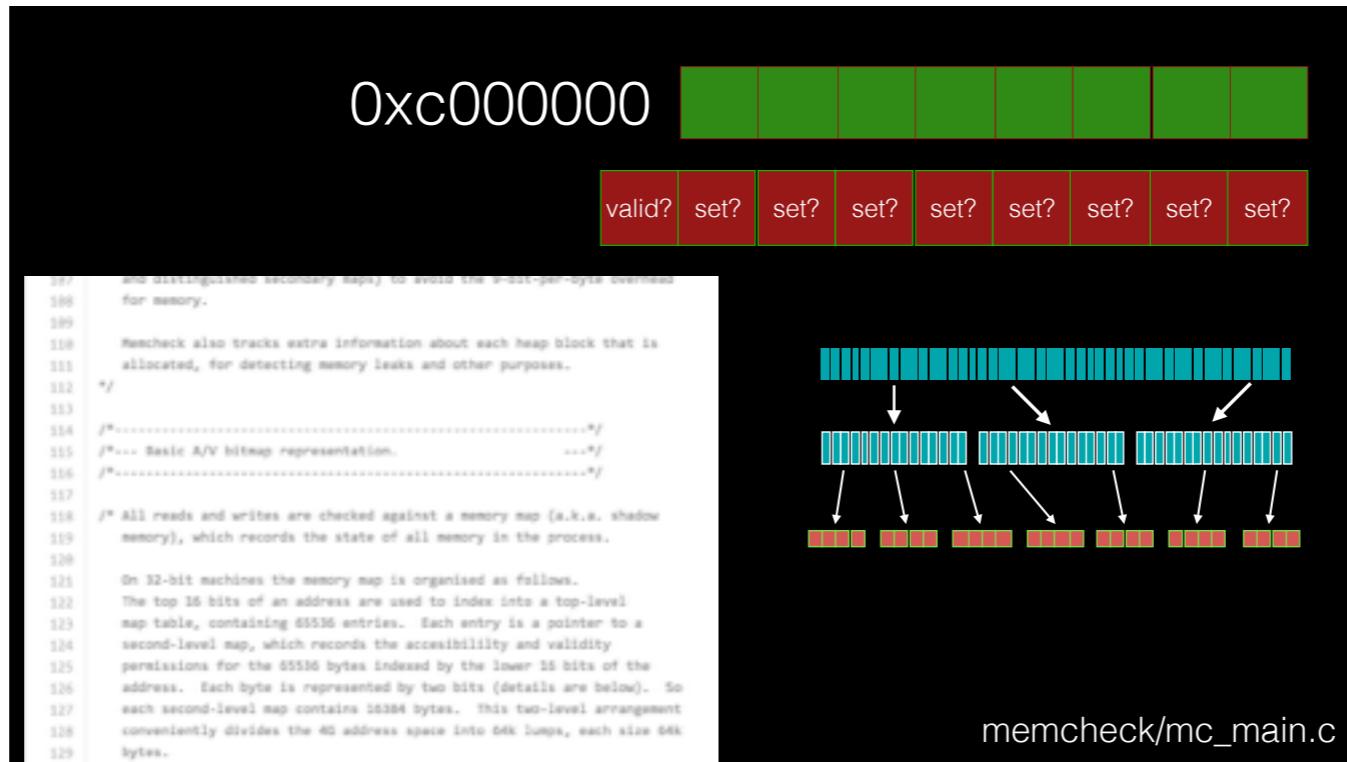
So what would an example of shadow memory look like? So let's imagine that we're building a tool that checks that we have always written to memory before reading from it. This is one of the things that Valgrind's memcheck plugin does. So memcheck's shadow state is, conceptually,



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
nicholas.nethercote@nicta.com.au

Julian Seward
OpenWrt.org, Cambridge, UK
jseward@openwrt.org

Abstract

Dynamic binary instrumentation (DBI) frameworks make it easy to build dynamic binary analysis (DBA) tools such as monitors and profilers. Much of the focus on DBI frameworks has been on performance. Little attention has been paid to their capabilities. This paper describes Valgrind, a DBI framework.

In this paper we describe Valgrind, a DBI framework designed for building heavyweight DBA tools. We focus on its unique cap-
abilities, which include: (1) support for many different kinds of values and efficient multi-threaded DBA methods, which requires a tool to shadow every register and memory value with another value that matches the original access; (2) support for value constraints that state that shadowed values from other threads match the state of their targets; (3) lightweight tools built with Valgrind that automatically analyze for bugs can be used to build static analyzers; (4) support for both shared memory and message-passing APIs in build with other DBI frameworks such as Pin and DyninstAPI.

Categories and Subject Descriptors: D.2.2 [Software Engineering]: Testing and Debugging—debugging aids, monitors; D.4.2 [Programming Languages]: Processors—instrumentation

General Terms: Design, Performance, Experimentation

Keywords: Valgrind, Monitors, dynamic binary instrumentation, dynamic binary analysis, shadow values

1. Introduction

Valgrind is a dynamic binary instrumentation (DBI) framework for Linux and Mac OS X. It is a framework, not a tool. This paper describes how it works, and how it differs from other frameworks.

1.1 Previous Binary Analysis and Instrumentation

Many environments are program analysis tools, such as static checkers and provers, to improve the quality of their software. Dynamic binary analysis (DBA) tools are similar, but they analyze programs at run-time at the level of machine code. Other tools are often implemented using dynamic binary instrumentation, such as Valgrind, or as a component of a larger suite of tools that analyze programs at run-time. This is convenient for users,

as no recompilation (such as disassembling or re-linking) is needed. After all, DBI frameworks make it easy to build DBA tools without requiring extensive code. Several generic DBI frameworks exist, such as Pin [13], DyninstAPI [14], and Valgrind [15, 16]. They have strengths and weaknesses. Pin is a general-purpose DBI, while Valgrind is built specifically for memory analysis. Both are excellent for writing tools that plug into the three systems.

The performance of DBI frameworks has been studied closely [3, 2, 10]. Low overhead is important for DBI frameworks, but so is efficiency. Some tools have both goals. 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 (Heavyweight DBA)

One interesting group of DBI tools are those that use shadow values. These tools implement mostly in user-space, many register and memory values with another value that says something about it. We call these shadow values. Consider the following monitoring task. The monitor wants to know if the value of a register is zero. It does not care what the value is, just that it is zero. It also wants to know that shadow values can be used in a wide variety of ways, and (2) are powerful and interesting.

For example, shadow values can track which file values are modified (i.e., unmodified, or derived from modified values) and use these derived properties over a modified value. It is used by the Valgrind’s massif memory profiler, and is probably the most well-known DBI tool in use.

Valgrind [15] tracks which file values are tainted (i.e., from an untrusted source, or derived from untrusted values). However, Valgrind [15] and Massif [16] do not track shadow values. However, Pin [13] and Dyninst [14] do.

Massif [16] and Pin [13] have monitoring, and Valgrind [15] which has monitoring, but neither of them tracks shadow values. Valgrind [15] tracks shadow values by type (implementation from open source), and can track shadow values for each type. This makes Valgrind [15] appropriate for a wider range of applications.

Dyninst [14] tracks shadow values for each thread, and Valgrind [15] tracks shadow values for each value. It is a good example of how much information about recent usage is available by public request.

Valgrind [15] tracks which word values are array pointers, and uses this to detect buffer overflows.

Valgrind [15] tracks which word values are heap pointers, and uses this to detect heap overflows.

Valgrind [15] tracks which word values are stack pointers, and uses this to detect stack overflows.

Valgrind [15] tracks which word values are function pointers, and uses this to detect function overflows.

Valgrind [15] tracks which word values are file pointers, and uses this to detect file overflows.

Valgrind [15] tracks which word values are socket pointers, and uses this to detect socket overflows.

Valgrind [15] tracks which word values are shared pointers, and uses this to detect shared pointer overflows.

Valgrind [15] tracks which word values are mutex pointers, and uses this to detect mutex overflows.

Valgrind [15] tracks which word values are condition pointers, and uses this to detect condition overflows.

Valgrind [15] tracks which word values are semaphores, and uses this to detect semaphore overflows.

Valgrind [15] tracks which word values are file descriptors, and uses this to detect file descriptor overflows.

Valgrind [15] tracks which word values are network sockets, and uses this to detect network socket overflows.

Valgrind [15] tracks which word values are shared memory pointers, and uses this to detect shared memory pointer overflows.

Valgrind [15] tracks which word values are memory mapped files, and uses this to detect memory mapped file overflows.

Valgrind [15] tracks which word values are shared memory regions, and uses this to detect shared memory region overflows.

Valgrind [15] tracks which word values are shared memory locks, and uses this to detect shared memory lock overflows.

Valgrind [15] tracks which word values are shared memory notifications, and uses this to detect shared memory notification overflows.

Valgrind [15] tracks which word values are shared memory semaphores, and uses this to detect shared memory semaphore overflows.

Valgrind [15] tracks which word values are shared memory mutexes, and uses this to detect shared memory mutex overflows.

Valgrind [15] tracks which word values are shared memory conditions, and uses this to detect shared memory condition overflows.

Valgrind [15] tracks which word values are shared memory regions, and uses this to detect shared memory region overflows.

Valgrind [15] tracks which word values are shared memory locks, and uses this to detect shared memory lock overflows.

Valgrind [15] tracks which word values are shared memory notifications, and uses this to detect shared memory notification overflows.

Valgrind [15] tracks which word values are shared memory semaphores, and uses this to detect shared memory semaphore overflows.

Valgrind [15] tracks which word values are shared memory mutexes, and uses this to detect shared memory mutex overflows.

Valgrind [15] tracks which word values are shared memory conditions, and uses this to detect shared memory condition overflows.

Valgrind [15] tracks which word values are shared memory regions, and uses this to detect shared memory region overflows.

Valgrind [15] tracks which word values are shared memory locks, and uses this to detect shared memory lock overflows.

Valgrind [15] tracks which word values are shared memory notifications, and uses this to detect shared memory notification overflows.

Valgrind [15] tracks which word values are shared memory semaphores, and uses this to detect shared memory semaphore overflows.

Valgrind [15] tracks which word values are shared memory mutexes, and uses this to detect shared memory mutex overflows.

Valgrind [15] tracks which word values are shared memory conditions, and uses this to detect shared memory condition overflows.

Valgrind [15] tracks which word values are shared memory regions, and uses this to detect shared memory region overflows.

Valgrind [15] tracks which word values are shared memory locks, and uses this to detect shared memory lock overflows.

Valgrind [15] tracks which word values are shared memory notifications, and uses this to detect shared memory notification overflows.

Valgrind [15] tracks which word values are shared memory semaphores, and uses this to detect shared memory semaphore overflows.

Valgrind [15] tracks which word values are shared memory mutexes, and uses this to detect shared memory mutex overflows.

Valgrind [15] tracks which word values are shared memory conditions, and uses this to detect shared memory condition overflows.

Valgrind [15] tracks which word values are shared memory regions, and uses this to detect shared memory region overflows.

Valgrind [15] tracks which word values are shared memory locks, and uses this to detect shared memory lock overflows.

Valgrind [15] tracks which word values are shared memory notifications, and uses this to detect shared memory notification overflows.

Valgrind [15] tracks which word values are shared memory semaphores, and uses this to detect shared memory semaphore overflows.

Valgrind [15] tracks which word values are shared memory mutexes, and uses this to detect shared memory mutex overflows.

Valgrind [15] tracks which word values are shared memory conditions, and uses this to detect shared memory condition overflows.

Valgrind [15] tracks which word values are shared memory regions, and uses this to detect shared memory region overflows.

Valgrind [15] tracks which word values are shared memory locks, and uses this to detect shared memory lock overflows.

Valgrind [15] tracks which word values are shared memory notifications, and uses this to detect shared memory notification overflows.

Valgrind [15] tracks which word values are shared memory semaphores, and uses this to detect shared memory semaphore overflows.

Valgrind [15] tracks which word values are shared memory mutexes, and uses this to detect shared memory mutex overflows.

Valgrind [15] tracks which word values are shared memory conditions, and uses this to detect shared memory condition overflows.

Valgrind [15] tracks which word values are shared memory regions, and uses this to detect shared memory region overflows.

Valgrind [15] tracks which word values are shared memory locks, and uses this to detect shared memory lock overflows.

Valgrind [15] tracks which word values are shared memory notifications, and uses this to detect shared memory notification overflows.

Valgrind [15] tracks which word values are shared memory semaphores, and uses this to detect shared memory semaphore overflows.

Valgrind [15] tracks which word values are shared memory mutexes, and uses this to detect shared memory mutex overflows.

Valgrind [15] tracks which word values are shared memory conditions, and uses this to detect shared memory condition overflows.

Valgrind [15] tracks which word values are shared memory regions, and uses this to detect shared memory region overflows.

Valgrind [15] tracks which word values are shared memory locks, and uses this to detect shared memory lock overflows.

Valgrind [15] tracks which word values are shared memory notifications, and uses this to detect shared memory notification overflows.

Valgrind [15] tracks which word values are shared memory semaphores, and uses this to detect shared memory semaphore overflows.

Valgrind [15] tracks which word values are shared memory mutexes, and uses this to detect shared memory mutex overflows.

Valgrind [15] tracks which word values are shared memory conditions, and uses this to detect shared memory condition overflows.

Valgrind [15] tracks which word values are shared memory regions, and uses this to detect shared memory region overflows.

Valgrind [15] tracks which word values are shared memory locks, and uses this to detect shared memory lock overflows.

Valgrind [15] tracks which word values are shared memory notifications, and uses this to detect shared memory notification overflows.

Valgrind [15] tracks which word values are shared memory semaphores, and uses this to detect shared memory semaphore overflows.

Valgrind [15] tracks which word values are shared memory mutexes, and uses this to detect shared memory mutex overflows.

Valgrind [15] tracks which word values are shared memory conditions, and uses this to detect shared memory condition overflows.

Valgrind [15] tracks which word values are shared memory regions, and uses this to detect shared memory region overflows.

Valgrind [15] tracks which word values are shared memory locks, and uses this to detect shared memory lock overflows.

Valgrind [15] tracks which word values are shared memory notifications, and uses this to detect shared memory notification overflows.

Valgrind [15] tracks which word values are shared memory semaphores, and uses this to detect shared memory semaphore overflows.

Valgrind [15] tracks which word values are shared memory mutexes, and uses this to detect shared memory mutex overflows.

Valgrind [15] tracks which word values are shared memory conditions, and uses this to detect shared memory condition overflows.

Valgrind [15] tracks which word values are shared memory regions, and uses this to detect shared memory region overflows.

Valgrind [15] tracks which word values are shared memory locks, and uses this to detect shared memory lock overflows.

Valgrind [15] tracks which word values are shared memory notifications, and uses this to detect shared memory notification overflows.

Valgrind [15] tracks which word values are shared memory semaphores, and uses this to detect shared memory semaphore overflows.

Valgrind [15] tracks which word values are shared memory mutexes, and uses this to detect shared memory mutex overflows.

Valgrind [15] tracks which word values are shared memory conditions, and uses this to detect shared memory condition overflows.

Valgrind [15] tracks which word values are shared memory regions, and uses this to detect shared memory region overflows.

Valgrind [15] tracks which word values are shared memory locks, and uses this to detect shared memory lock overflows.

Valgrind [15] tracks which word values are shared memory notifications, and uses this to detect shared memory notification overflows.

Valgrind [15] tracks which word values are shared memory semaphores, and uses this to detect shared memory semaphore overflows.

Valgrind [15] tracks which word values are shared memory mutexes, and uses this to detect shared memory mutex overflows.

Valgrind [15] tracks which word values are shared memory conditions, and uses this to detect shared memory condition overflows.

Valgrind [15] tracks which word values are shared memory regions, and uses this to detect shared memory region overflows.

Valgrind [15] tracks which word values are shared memory locks, and uses this to detect shared memory lock overflows.

Valgrind [15] tracks which word values are shared memory notifications, and uses this to detect shared memory notification overflows.

Valgrind [15] tracks which word values are shared memory semaphores, and uses this to detect shared memory semaphore overflows.

Valgrind [15] tracks which word values are shared memory mutexes, and uses this to detect shared memory mutex overflows.

Valgrind [15] tracks which word values are shared memory conditions, and uses this to detect shared memory condition overflows.

Valgrind [15] tracks which word values are shared memory regions, and uses this to detect shared memory region overflows.

Valgrind [15] tracks which word values are shared memory locks, and uses this to detect shared memory lock overflows.

Valgrind [15] tracks which word values are shared memory notifications, and uses this to detect shared memory notification overflows.

Valgrind [15] tracks which word values are shared memory semaphores, and uses this to detect shared memory semaphore overflows.

Valgrind [15] tracks which word values are shared memory mutexes, and uses this to detect shared memory mutex overflows.

Valgrind [15] tracks which word values are shared memory conditions, and uses this to detect shared memory condition overflows.

Valgrind [15] tracks which word values are shared memory regions, and uses this to detect shared memory region overflows.

Valgrind [15] tracks which word values are shared memory locks, and uses this to detect shared memory lock overflows.

Valgrind [15] tracks which word values are shared memory notifications, and uses this to detect shared memory notification overflows.

Valgrind [15] tracks which word values are shared memory semaphores, and uses this to detect shared memory semaphore overflows.

Valgrind [15] tracks which word values are shared memory mutexes, and uses this to detect shared memory mutex overflows.

Valgrind [15] tracks which word values are shared memory conditions, and uses this to detect shared memory condition overflows.

Valgrind [15] tracks which word values are shared memory regions, and uses this to detect shared memory region overflows.

Valgrind [15] tracks which word values are shared memory locks, and uses this to detect shared memory lock overflows.

Valgrind [15] tracks which word values are shared memory notifications, and uses this to detect shared memory notification overflows.

Valgrind [15] tracks which word values are shared memory semaphores, and uses this to detect shared memory semaphore overflows.

Valgrind [15] tracks which word values are shared memory mutexes, and uses this to detect shared memory mutex overflows.

Valgrind [15] tracks which word values are shared memory conditions, and uses this to detect shared memory condition overflows.

Valgrind [15] tracks which word values are shared memory regions, and uses this to detect shared memory region overflows.

Valgrind [15] tracks which word values are shared memory locks, and uses this to detect shared memory lock overflows.

Valgrind [15] tracks which word values are shared memory notifications, and uses this to detect shared memory notification overflows.

Valgrind [15] tracks which word values are shared memory semaphores, and uses this to detect shared memory semaphore overflows.

Valgrind [15] tracks which word values are shared memory mutexes, and uses this to detect shared memory mutex overflows.

Valgrind [15] tracks which word values are shared memory conditions, and uses this to detect shared memory condition overflows.

Valgrind [15] tracks which word values are shared memory regions, and uses this to detect shared memory region overflows.</p

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

PLDI '07

Table 3. Performance of three weighted models on MNIST-175 (2008). Columns 1 gives the program names, longer programs are listed below shorter ones. Programs in Column 2 give the entire execution time, and programs in Column 3 give the time spent in the main loop.

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.

“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.



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.



“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 resynthesize” 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,bpf,taig,kcchrist,mcrossen)@cs.stanford.edu
Stanford University Department of Computer Science

Abstract

(Abstract) Tracing the lifetime (i.e., propagation and duration of ownership) of sensitive data (e.g., passwords) is an important and well accepted practice in secure software development. Unfortunately, there are no current tools available for automatically tracking data lifetime and very little information available on the practice of today's software with respect to data lifetime.

We describe a system that has developed over several years through iterative refinement and reuse of open source code. This article details the system's design for "tainting" at the hardware level. Tainting information is then propagated across operating system, language, and application layers, allowing analysis of sensitive data handling at a whole system level.

We have used this tool 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. The analysis reveals that these applications use the lifetime of sensitive data to propagate sensitive data through the lifetime of sensitive data they handle. Logging password and other sensitive data scattered throughout user and kernel memory. We show how simple and practical changes can greatly reduce sensitive data lifetime in these applications.

I Introduction

Examining sensitive data lifetime can lead valuable insights into the security of software systems. When analyzing data lifetimes we are concerned with two primary questions: how long does sensitive data live in an operating system, library, application? (longer data is more interesting alive (i.e., in an accessible form in memory or persistent storage) and where components propagate data (e.g., buffers, log files, other components).

A data lifetime measures as data the likelihood of exposure to an attack. Data can become exposed by a number of different mechanisms: 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/history, session management, crash dump, or crash reporting [5], interactive core 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, application libraries, program buffers, memory, loggers, 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 programming language.

To overcome these limitations we have developed a tool based on whole-system simulation called *TaintBench*, 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 *TaintBench* by running the entire software stack under simulation. The system is simulated 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.

TaintBench provides two flags whenever their respective values in hardware are modified in an operation. These values are tracked throughout the system as it flows through kernel device drivers, user-level GPU widgets, application buffers, etc. Tainting is introduced when sensitive data enters the system, such as when a password is read from the keyboard driver, or application reads a particular data file, etc.

The next section details our findings on the lifetime

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 Gorfinkel, Kevin Christopher, Mendel Rosenblum
(jchow,bpf,tdg,kchristopher,mroenbl)@cs.stanford.edu
Stanford University Department of Computer Science

Abstract

(Briefly) Tracing the lifetime (i.e., propagation and duration of ownership) of sensitive data (e.g., passwords) is an important and well accepted practice in secure software development. Unfortunately, there are no current tools available for automatically tracing data lifetime, which are little information available on the practice of today's software code with respect to data lifetime.

We describe a system we have developed for analyzing sensitive data lifetime at the hardware level. TaintBochs is a "tainting" of 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, credit card numbers, etc. We studied how the application asserts that their application and the operating system are handling sensitive data correctly. We found the lifetime of sensitive data may leak through password and other sensitive data handled throughout user and kernel memory. We show how simple and practical changes can greatly reduce sensitive data lifetime in these applications.

I Introduction

Examining sensitive data lifetime can lead valuable insight into the security of software systems. When analyzing data lifetimes we are concerned with two primary issues: how long does sensitive data remain in the system, and what components propagate data? (i.e., in an assembly line of 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 sensitive data being exposed to an attacker. Data can leave by one of many paths: going across to system memory or to persistent storage (e.g., swap space) to which data has leaked. Continue data handling also increases the risk of data exposure via interaction with buffers such as logging, command/history, session management, crash dump, or crash reporting [5], interactive core 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, application libraries, program buffers, memory, swap space, 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 programming 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 on a simulated hardware platform. The system inside a simulated environment. Every byte of system memory, device state, and relevant processor state is tagged with a taint status. Tag. Data is "tainted" if it is considered sensitive.

TaintBochs propagates taint flags whenever their value is updated in hardware, or modified in an application. Thus, tainted data is tracked throughout its lifetime 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 driver, or application reads a particular data file.

We used TaintBochs to analyze 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. In fact, many applications were found to be exposing sensitive data to an attacker by leaving sensitive data within being cleared of their memory after being copied to the heap indefinitely. Sensitive data was left in cleared memory for indeterminate periods without good reason, and unnecessary applications issued unnecessary 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

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 Gorfinkel, Kevin Christopher, Mendel Rosenblum
(jchow,bpf,tdg,kchristopher,mroenbl)@cs.stanford.edu
Stanford University Department of Computer Science

Abstract

Data lifetime (i.e., the propagation and duration of ownership) of sensitive data (e.g., password) is an important and well accepted practice in secure software development. Unfortunately, there are no current tools available for automatically tracking data lifetime and very little information available on the practice of today's software with respect to data lifetime.

We describe a system that we have developed for analyzing data lifetime in several large real-world applications. Specifically, our tool tracks data for "writing" at the hardware level. Tracing 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 our tool 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 analysis reveals that these applications use the lifetime of sensitive data to their advantage. For example, in the lifetime of sensitive data they securely store passwords and other sensitive data scattered throughout user and kernel memory. We show how simple and practical changes can greatly reduce sensitive data lifetime in these applications.

I Introduction

Examining sensitive data lifetime can lead valuable insight into the security of software systems. When analyzing data lifetimes we are concerned with two primary issues: how long does sensitive material live in an operating system, library, application? (longer data is more “living” alive (i.e. in an accessible form in memory or persistent storage) and where components propagate data (e.g. buffers, log files, other components).

An data lifetime increases as does the likelihood of exposure to an attack. Data can leak by virtue 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/history, session management, crash dump or crash reporting [5], interactive core 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, application programs, libraries, languages, compilers, interpreters, 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 programming language.

To overcome these limitations we have developed a tool based on whole-system simulation called SimBochs, 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 SimBochs by running the entire software stack on a simulated processor running in a simulated environment. Every byte of system memory, device state, and relevant processor state is tagged with a total status flag. Data is “written” if it is considered sensitive.

SimBochs provides fast flags whenever their value is valid in hardware and simulated in an application. Thus, valid data is tracked throughout the system as it flows through kernel, device drivers, user-level GUI widgets, application buffers, etc. Tracing is introduced when sensitive data enters the system, such as when a password is read from the keyboard driver, an application reads a particular data file, etc.

Our initial SimBochs is for exploring the lifetime

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 Gorfinkel, Kevin Christopher, Mendel Rosenblum
(jchow,bpf,tdg,kchristopher,mroenbl)@cs.stanford.edu
Stanford University Department of Computer Science

Abstract

(Briefly) Understanding 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 tools available for automatically analyzing data lifetime and very little information available on the quality of today's approaches with respect to their lifetime.

We describe a system through which we have developed a methodology for automatically understanding sensitive data lifetime. TaintBochs models sensitive data by "tainting" it at the hardware level. Tracing information is then propagated across operating system, language, and application layers, allowing 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. in a daily basis. Our analysis reveals that these applications use the lifetime of sensitive data to their advantage. For example, in the lifetime of sensitive data they identify logging password and other sensitive data scattered throughout user and kernel memory. We show how simple and practical changes can greatly reduce sensitive data lifetime in these applications.

I Introduction

Examining sensitive data lifetime can lend valuable insight into the security of software systems. When analyzing data lifetimes we are concerned with two primary issues: how long does sensitive information stay in the system, and what happens to the data during its lifetime? The lifetime of sensitive data is often measured in terms of how long it is propagating 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 attack. Data can leak by virtue 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/history, session management, crash dumps, or crash reporting [5], interactive core reporting, etc.

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

We have augmented TaintBuchi with three capabilities to produce TaintBuchi+. First, we provide the ability to track the propagation of sensitive data through the system at a hardware level, i.e., tainting. Second, we have added logging capabilities that allow system state such as memory and registers at any given time during a system's execution to be analyzed. Finally, we developed an analysis framework that allows information about CPU intervals, along with information for the software that is running, to be utilized as an integrated feature to allow easy interpretation of tainting information. This allows us to trace tainted data to an exact program variable in an application (or the kernel) in the guest, and easily propagate tainted-to-an-exact-source the address where the tainted data originated.

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

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

3.1 Hardware Level Tainting

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

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

By a conservative propagation, we mean propagating sensitive data, although some data may be inherently tainted. Propagation would take significant additional effort, but we have not yet encountered a situation where this would noticeably aid our analysis.

For simplicity, TaintBuchi only maintains shadow memory for the guest's main memory and the 16-32bit device state (e.g., CPU registers, memory, and registers, ROM (e.g., BIOS), PCI) registers, and flags are disregarded, as to chip-set and I/O device state. Adding the necessary tracking for other processor or I/O device state (e.g., disk, frame buffer) would be quite straightforward, but the current implementation is sufficient for many kinds of useful analysis. We are not seriously concerned about the guest's ability to launder tainted data through various data structures, but we are concerned that the software under analysis is not intentionally malicious.

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

In the trivial case where data is simply copied, we perform the same operation in the address space of shadow memory. So, if the assignment $A \leftarrow B$ occurs on normal memory, then $A - B$ is also contained on shadow memory. Correspondingly, if B was tainted then A is now also tainted, and if B was not tainted, it is now also not tainted.

The answer is less straightforward when an instruction produces a new value based on a set of inputs. In each case, our simulation must decide on whether and how to update the instruction's output(s). One obvious choice balances the desire to preserve as many possibly interesting states against the need to make the question more tractable by limiting too much data or operations involving this. This roughly corresponds to the filter negatives in. Note positive trade-offs made in different analysis tools. As we will see, it is in general impossible to achieve the last goal perfectly, so some compromise must be made.

Processor instructions typically produce outputs that are some function of their inputs. Our basic propagation rule is to copy any byte of any input register to a tainted memory location if any part of that byte is tainted, even if the rest of the byte is not. This policy is clearly conservative and errs on the side of making too much. Interestingly though, with the exception of cases noted below, we have not yet encountered any obviously spurious output resulting from our 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

We have augmented Tandembus with three capabilities to produce Tandembus+. First, we provide the ability to track the propagation of sensitive data through the system at a hardware level, i.e. tainting. Second, we have added logging capability that allows system state such as memory and registers at any given time during a system's execution to be captured. Finally, we developed an analysis framework that allows information about CPU intervals, along with information for the software that is running, to be utilized as an integrated indicator to allow easy interpretation of tainted information. This allows us to trace tainted data to its exact program source in an application (or the kernel) in the guest, and easily propagate tainted-to-an-exact-source the audience.

The basic usage model consists of two phases. First, we run a simulation in which sensitive data (e.g. coming from the keyboard, network, etc.) is identified as tainted. The workflow consists of normal user interactions, e.g. logging into a website via a browser. In the second phase, the tainted data is compared with the user input. This allows us to determine if there are open-coded queries about the simulation, such as where tainted data came from, where it was stored, how it was propagated, etc.

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

3.1 Hardware Level Tainting

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

Sensitive Memory To track the location of sensitive data in Tandembus+, we added another memory, set of registers, etc., called a shadow memory. The shadow memory tracks tainted status of every byte in the system. Every operation performed on machine state by the processor or device driver is a procedure to be performed in shadow memory. e.g. register A is copied from register B to location B copies the value in the shadow register A to be copied to shadow location B. Thus to determine if a byte is tainted we need only look at the corresponding location in shadow memory.

By a *tainting operation*, i.e. we do not track mask of sensitive data, although some data may be inherently tainted. By granularity would take significant additional effort, but we have not yet determined if this is the case when this would actually aid our analysis.

For simplicity, Tandembus+ only maintains shadow memory for the guest's main memory and the 16-32bit address space. It also tracks memory writes, reads, and regions (SMM (e.g. MMIO, MMU) regions, and flags are disregarded), but the current implementation is sufficient for many kinds of useful analyses. We are not seriously concerned about the guest's ability to launder tainted data through the use of flushing registers, for example. But the assumption is that the software under analysis is not intentionally malicious.

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

In the trivial case where data is simply copied, we perform the same operation in the address space of shadow memory. So, if the assignment $A \leftarrow B$ occurs on normal memory, then $A - B$ is also cleaned on shadow memory. Consequently, if B was tainted then it is now also tainted, and if B was not tainted, it is now also not tainted.

The answer is less straightforward when an instruction produces a new value based on a set of inputs. In each case, we must decide on whether and how to update the instruction's output(s). One obvious choice balances the desire to preserve as many possibly innocent values against the need to make sure operations resulting from tainted data do not corrupt other values. This roughly corresponds to the false negatives in. Note positive trade-offs made in collecting analysis results. As we will see, it is in general impossible to achieve the best of both worlds, so some compromise must be made.

Processor instructions typically produce outputs that are some function of their inputs. Our basic propagation policy is to copy any bit of any input value to the output, even if only part of the output is affected. This policy is clearly conservative and errs on the side of being too much. Interestingly though, with the exception of cases noted below, we have not yet enumerated any obviously spurious output resulting from our policy.

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

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

set, etc.

We have augmented Taintblocks with three capabilities to produce Taintblocks+. First, we provide the ability to track the propagation of sensitive data through the system at a hardware level, i.e. tainting. Second, we have added logging capabilities that allow system state such as memory and registers at any given time during a system's execution to be captured. Finally, we developed an analysis framework that allows information about CPU instructions, along with information for the software that is running, to be utilized in an integrated fashion to allow easy interpretation of tainted information. This allows us to trace tainted data to an exact program variable in an application (or the kernel) in the guest, and easily propagate tainted-to-an-exact-source the audience.

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

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

3.1 Hardware Level Tainting

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

Shadow Memory To track the location of sensitive data in Taintblocks, we added another memory, set of registers, etc., called a shadow memory. The shadow memory tracks tainted status of every byte in the system. Every operation performed on machine state by the processor or device driver is a programmatic write to be recorded in shadow memory. e.g. register A is from register B so location B contains the same as the shadow register A so copied to shadow location B. Thus to determine if a byte is tainted we need only look at the corresponding location in shadow memory.

we are tracking sensitive data as a very granularly as is reasonable possible, i.e. we do not track much of sensitive data, although some data may be reasonably tainted. By granularity would take significant additional effort, but we have not yet determined if this is worth the effort.

For simplicity, Taintblocks only maintains shadow memory for the guest's main memory and the 16-32bit address space of the guest. Cache, memory, and registers (e.g. MMX, SIMD) registers, and flags are disregarded, as is chip-set and I/O device state. Adding the necessary tracking for other processor or I/O device state (e.g. disk, frame buffer) would be quite straightforward, but the current implementation is sufficient for many kinds of useful analysis. We are not seriously concerned about the guest's ability to launder tainted data through the various tainting registers, but assume that our interpretation is the software under analysis is reasonably malicious.

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

In the trivial case where data is simply copied, we perform the same operation in the address space of shadow memory. So, if the assignment $A \leftarrow B$ occurs on normal memory, then $A - B$ is also contained in shadow memory. Correspondingly, if B was tainted then A is now also tainted, and if B was not tainted, it is now also not tainted.

The answer is less straightforward when an instruction produces a new value based on a set of inputs. In each case, our simulation must decide on whether and how to taint the instruction's output(s). One obvious choice balances the desire to preserve any possibly interesting state against the need to make the question more tractable by reducing too much data or ignoring certain details. This roughly corresponds to the false negatives in. Since positive trade-offs made in earlier analysis make. As we will see, it is in general impossible to achieve the last goal perfectly, so some compromise must be made.

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

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.

We have augmented TaintBuchi with three capabilities to produce TaintBuchi+. First, we provide the ability to track the propagation of sensitive data through the system at a hardware level, i.e. tainting. Second, we have added logging capability that allows system state such as memory and registers at any given time during a system's execution to be captured. Finally, we developed an analysis framework that allows information about CPU intervals, along with information for the software that is running, to be utilized as an integrated indicator to allow easy interpretation of tainted information. This allows us to trace tainted data as it moves through memory to an application (or the kernel) in the guest, and easily propagate tainted-to-an-exec source the audience.

The basic usage model consists of two phases. First, we run a simulation in which sensitive data (e.g. coming from the keyboard, network, etc.) is identified as tainted. The workflow consists of normal user interactions, e.g. logging into a website via a browser. In the second phase, the tainted data is analyzed with the analysis framework. This allows us to trace tainted data from its original source through the simulation, such as where tainted data came from, where it was stored, how it was propagated, etc.

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

3.1 Hardware Level Tainting

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

Sensitive Memory To track the location of sensitive data in TaintBuchi, we added another memory, set of registers, etc., called a shadow memory. The shadow memory tracks tainted status of every byte in the system. Every operation performed on machine state by the processor or device driver is a programmatic write to be recorded in shadow memory. e.g. if register A is from register B to location B contains the same as the shadow register A to be copied to shadow location B. Thus to determine if a byte is tainted we need only look at the corresponding location in shadow memory.

By a conservative propagation, i.e. we do not track mask of sensitive data, although some data may be potentially tainted. By granularity we take minimal additional effort, but we have not yet considered a situation where this would noticeably aid our analysis.

For simplicity, TaintBuchi only maintains shadow memory for the guest's main memory and the 16-32bit address space. It does not track memory writes, reads, and regions (SMM (e.g. MMIO), PCIe regions, and flags are disregarded), as the current implementation is sufficient for memory tracking for other processor or I/O device state (e.g. disk, frame buffer) would be quite straightforward, but the current implementation is sufficient for many kinds of useful analysis. We are not seriously concerned about the guest's ability to launder tainted data through the use of flushing registers. For most of our experiments in this software under analysis is intentionally malicious.

Propagation Policy We must decide how operations in the system should affect shadow state. If we register A and B are tainted, and one of them is tainted, is the register where the result is stored also tainted? Is the register where the result is stored also tainted? We will call the collection of policies that decide this as the propagation policy.

In the trivial case where data is simply copied, we perform the same operation in the address space of shadow memory. So, if the assignment $A \leftarrow B$ occurs on normal memory, then $A - B$ is also contained in shadow memory. Consequently, if B was tainted then it is now also tainted, and if B was not tainted, it is now also not tainted.

The answer is less straightforward when an instruction produces a new value based on a set of inputs. In such cases, we must first decide on whether and how to update the instruction's output(s). One obvious choice balances the desire to preserve any possibly interesting state against the need to make the question more tractable by taking too much data or memory mapping. This roughly corresponds to the false negatives in. Note positive trade-offs made in earlier analysis tools. As we will see, it is in general impossible to achieve the best of both worlds, so some compromise must be made.

Processor instructions typically produce outputs that are some function of their inputs. One basic propagation rule is to copy any item of any input value to the output, that is, if g is a general processor instruction, and g takes n inputs and m outputs, then g is clearly conservative and sets on the side of being too strict. Interestingly though, with the exception of cases noted below, we have not yet encountered any obviously spurious output resulting from our analysis.

“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

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

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

This situation arises constantly. For example, Linux routinely merges keyboard device data through a hook-up to before sending keystrokes to user programs. Thus, user programs never directly see the data itself from the keyboard device, only the information indicates they index in the kernel's key mapping table.

Clearly this is not what we want, so we augmented our propagation policy to handle tainted indexes (i.e., tainted pointers) with the following rule: if any type of any input value that is involved in the address computation of a memory operation is tainted, then the output is tainted, regardless of the location of the tainted pointer. This is a rather broad heuristic.

▪ **Constant Functions.** Tainted values are sometimes used in computations that always produce the same result. We call such computations constant functions.

An example of such a computation might be the function `sin(12)` that computes out a regular, well-known, ... value. After all, the value is always the same, right?

For our purposes, the output of constant functions never pose a security risk, even with tainted inputs, since the input values are not derived from the output. In the `sin(12)` example above, it is so low the odds that the propagation could find tainted tainted outputs. ...

In this same case, our propagation policy treats the output, and in the `sin(12)` case, our propagation policy does not save the output (since intermediate inputs are never considered tainted).

Clearly, our desire is to never save the output of constant functions. And while this can clearly be done, it is not always the best choice. Consider the following sequence (the `x1`, `x2`, ..., `xn`, `fn`, `fn`, `fn`, ..., `fn`) in general since the general case (of which the `x1`, `x2`, ..., `xn`-examples are merely degenerate instances) is an arbitrary sequence of instructions that ultimately compute a constant function. For example, assuming `x1` is initially tainted, the sequence:

`x1 = 0x40; 2 * x1 = 2 * 0x40`

`fn(x1); fn(2 * x1) = 0`

Always computes (both consistently) zero for `x1`, regardless of the original value of `x1`. By the time the instruction `fn` reaches the `x1`, it has no knowledge of whether its operands have the same value because of some deterministic computation or through simple chance, it must decide, therefore, to treat the `x1` as tainted.

One way to mitigate a variety of phenomena addressed in this problem, our approach takes advantage of the semantics of tainted values. For our research, we are interested in tainted data representing secrets like a user typed password. Therefore, we simply define by fiat that we are only interested in taint on non-zero values. That is, any value that is a constant propagation and output value never becomes tainted, since zero outputs are, by definition, uninteresting.

This simple heuristic has the consequence that constant functions producing nonzero values can still be tainted. This has not been a problem in practice because the vast majority of constant functions are trivial enough for the debugger over the network in a reasonable time. Moreover, tainted inputs find their way into a constant function even more rarely, because tainted memory generally represents a fairly small fraction of the guest's overall memory.

One-way Functions. Constant functions are an interesting class of functions that are one-way functions. These we call one-way functions. A one-way function is characterized by the fact that its input is not easily derived from its output. The problem with one-way functions is that tainted input values generally produce tainted outputs, just as they did for constant functions. But since the output value gives no precise information about the input value's value, it is generally unnecessary to flag each output as tainted from the viewpoint of analyzing information leakage, since no practical security risk exists.

A concrete example of this situation occurs in Linux, where keyboard input is used as a source of entropy for the system's random pool. [2] In this case, the pool is generated through various mixing functions, which include cryptographic hashes like SHA-1. Although derivations of the original keyboard input are used by the kernel when it extracts entropy from the pool, no practical information can be gleaned about the original keyboard input from looking at the random number outputs (at least, not easily).

Our system does not currently try to ensure tainted outputs resulting from one-way functions.

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

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

• **Lookup Tables.** Sometimes tainted values are used as indexes into a lookup table. Since the tainted value itself is not used in the final computation, only the lookup value it points to, the propagation policy presented earlier would not classify the output as tainted.

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

Clearly this is not what we want, so we augment our propagation policy to handle tainted addresses (i.e., tainted pointers) with the following rule: if any type of any input value that is involved in the address computation of a memory operation is tainted, then the output is tainted, regardless of the intermediate operations that it undergoes.

• **Constant Functions.** Tainted values are sometimes used in computations that always produce the same result. We call such computations constant functions.

An example of such a computation might be the function `sin(12)` from `math.h` that always returns the same value. After all, `sin(12)` is equivalent to `sin(12 + 2 * pi)`, which is equivalent to `sin(12)`.

For our purposes, the output of constant functions never pass a security check, even with tainted inputs, since the input values are not derived from the output. In the `sin(12)` example above, it is so low the optimizer will recognize that `sin(12)` is a constant value. ☺

In this same case, our propagation policy would ignore the output, and in the `sin(12)` case, our propagation policy does not see the output (since intermediate inputs are never considered tainted).

Clearly, one desire is to never see the output of constant functions. And while this can clearly be done by simply ignoring the output of constant functions (the `sin(12)`, `sin(12)`, etc. cases), this cannot be done in general since the general case (of which the `sin(12)` and `cos(12)` examples are merely degenerate instances) is an arbitrary sequence of instructions that ultimately compute a constant function. For example, assuming `x=0` is initially tainted, the sequence:

`sin(x); x = 1; sin(x) - x;`

032: 004c: 2 j 0444 = 27 0308

033: 004c: 0308 r 004a = 0

Above computes (backward-chasing) zero for `x=0`, regardless of the original value of `x=x`. By the time the instruction execution reaches the `sin()`, it has no knowledge of whether its operands have the same value because of some deterministic computation (though simple chance, it must decide, therefore, to take no action).

Our approach integrates a variety of techniques to address this problem. Our approach takes advantage of the semantics of tainted values. For our research, we are interested in tainted data representing secrets like a user typed password. Therefore, we simply define by fiat that we are only interested in taints on non-secret data. This way, we can say that tainted programs and tainted value never touch non-secret, since non-secret are, by definition, uninteresting.

This simple heuristic has the consequence that constant functions producing tainted values can still be tainted. This has not been a problem in practice because most constant functions are not tainted. This is the reason for the degeneracy over the constant in a taint tree. Moreover, tainted inputs find their way into a constant function even more rarely, because tainted memory generally represents a fairly small fraction of the program's overall memory.

• **One-way Functions.** Constant functions are an interesting class of functions that are not invertible. These functions are called one-way functions. A one-way function is characterized by the fact that its input is not easily derived from its output. The problem with one-way functions is that tainted input values generally produce tainted outputs, just as they did for constant functions. But since the tainted value given to process is not easily derived from the program's code, it is generally unnecessary to flag such data as tainted from the viewpoint of analyzing information leakage, since no practical security risk exists.

A concrete example of this situation occurs in Linux, where keyboard input is used as a source of entropy for the random number pool. [Linux] uses one-way functions to process the raw keyboard input, and it is generally unnecessary to flag such data as tainted from the viewpoint of analyzing information leakage, since no practical security risk exists.

Our system does not currently try to ensure tainted outputs resulting from one-way functions.

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

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

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

- ➊ shadow memory to track information flow
- ➋ 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 booting a Linux⁷ guest inside TaintBochs. We then logged in as an unprivileged user, and started X with the tzen 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 TaintBochs'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).

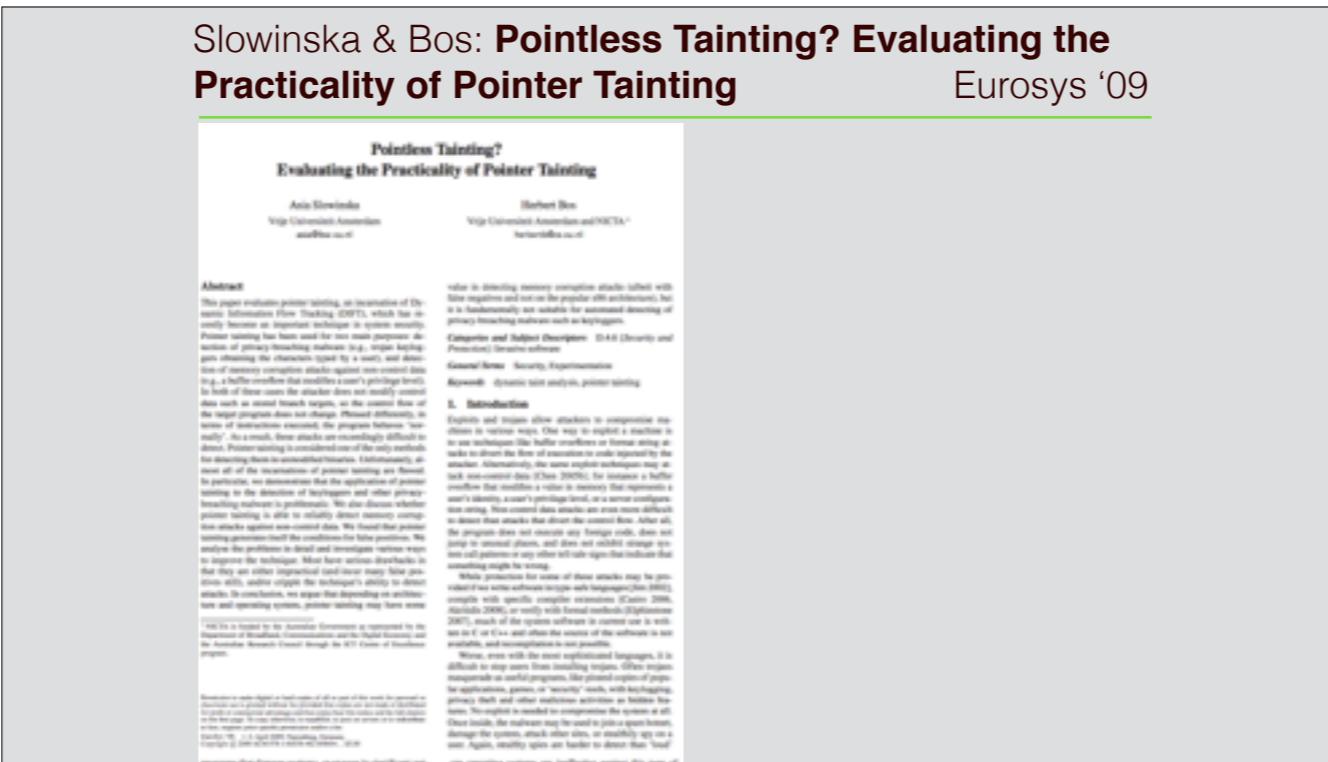
For our experiment, we began by booting a Linux⁷ guest inside TaintBochs. We then logged in as an unprivileged user, and started X with the tzen window manager. Inside X, we started Mozilla and brought up the webpage mail.yahoo.com, where we entered a user name and password in the login form. Before entering the password, we turned on TaintBochs'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_reancode` to keep track of the previous key press; the keyboard driver also has a similar variable `prev_reancode`.
- *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



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

Copyright © 2009 ACM, Inc. 0-9867708-0-5/09/03 \$15.00

programmes that damage systems, or engage in significant user activity. The keyloggers, installed by the user, may use legitimate API-to-APIs and move the characters that are typed in by the user (or data in files, buffers, or on the network). From a system's perspective, the malware is not doing anything "wrong".

In this paper, we distinguish between attacks that affect the control flow of a programme and those that do not. Control diversion typically means that a pointer in a programme is manipulated by an attacker so that when it is dereferenced, the programme acts according to the attacker's wishes. These types of attacks are often memory corruption attacks. On the other hand, memory corruption attacks against non-control-data and privacy breaking malware like keyloggers and sniffers. Memory corruption attacks against non-control-data manipulate data values that are used by the programme. For example, the value of a variable that represents a user's privilege level, or the length in bytes of a copy buffer. The attack itself does not lead to unusual code execution. Rather, it leads to elevated privileges, or unusual copies. The same is true for privacy-breaking malware like sniffers and regular keyloggers.

Pointer tainting as authorized or unauthorized. It is precisely these difficult-to-detect, usually non-control-diverting attacks that have been the focus of pointer tainting [Bink 2004]. At the same time, the focus of pointer tainting [Bink 2004] is on control-diverting attacks. We will discuss pointer tainting in more detail in later sections. Put simply, it suffices to define it as a form of dynamic information flow tracking [Bink 2004] which tracks the origin of data by way of a pointer to it in a manner analogous to its use in memory debugging. By monitoring the propagation of tainted data through the system (e.g., when tainted data is copied, but also when tainted pointers are dereferenced), we see whether any value derived from data from a tainted origin ends up in places where it should never appear. For example, if a programme has been compromised to track the propagation of user input data to ensure that untrusted and semi-trusted programs do not receive it [Tsai 2003], by implementing pointer tainting in hardware [Gill 2003], the assumption is that:

“...pointer tainting is very popular because (i) it can be applied to untrusted software without recompilation, and (ii) according to its advocates, it incurs hardly (if any) false positives, and (iii) it is assumed to be one of the only (if not the only) reliable techniques capable of detecting both memory corruption and pointer tainting attacks.” [Tsai 2003] Our paper argues that pointer tainting has become a unique and extremely valuable detection method especially due to its promised ability to detect non-control-diverting attacks. We illustrate earlier, semi-control-diverting attacks are more challenging than control-diverting attacks due to the fact that they are harder to detect. Control pointer mechanisms like address space randomization and stackguard [Blaauw 2005; Crouse 1998] present in several mod-

els. Again, memory spies are harder to detect than “bad” memory systems are ineffective against this type of attack. The main idea is to use all forms of system call monitoring [Purwo 2003; Olfert 2004]. As a result, some regular keyloggers have been active for years (often undetected). In one particularly worrying case, a keylogger has caused over 500,000 legal audits for online banking and e-commerce sites. The reason is that the memory corruption response of a successful non-control-diverting attack may be as severe as with a control-diverting attack. For instance, passwords obtained by a keylogger often give attackers full access to the machine. The issue is how to buffer overflows that modify a user's privilege level.

However, pointer tainting as our working as *admitted*, inspired by a series of publications about pointer tainting [Yin 2002; Feltz 2007; Dufour 2007; Yin 2008; Nekrasov 2008; Dufour 2009], several of which claim zero false positives, we tried to build a keylogger detector by means of pointer tainting. However, what we found is that the accuracy-reaching software detection, which is based on a large number of control and semi-control attacks. The false positives appear particularly hard to avoid. This is no easy fix. Further, we found that almost all existing applications of pointer tainting in detection of memory corruption attacks are also problematic, and none of them are suitable for the popular SSE architecture and Windows.

In this paper, we analyse the fundamental limitations of the method when applied to detection of privacy breaking malware, as well as the practical limitations in control and semi-control attacks. In contrast to previous efforts, we find out that the problem is that “filtering the memory footprint” is a simple solution to overcome the symptoms (rather than the root cause) related to false positives or false negatives.

Others have discussed issues with projects that use pointer tainting [Cohen 2003], and have also shown that pointer tainting is not a good technique [Tsai 2003]. To the best of our knowledge, nobody has investigated the technique in detail. Our study has shown that it does not work against keyloggers, and we are the first to report the complicated problems with the techniques that are hard to overcome. We are also the first to conclude the contributions of this paper are:

1. an in-depth analysis of the problem of pointer tainting and the reasons why pointer tainting is not able to detect memory spying or semi-control diversion, and is problematic in other forms also;
2. an analysis and evaluation of all known flaws in the problems that shows that they all have serious shortcomings.

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

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

programmes that damage systems, or engage in significant user activity. The majority, however, incurred by the user, may use legitimate API-to-APIs and move the pointers that are typed in by the user (or data in files, buffers, or on the network). From a system's perspective, the malware is not doing anything "wrong".

In fact, pointer tainting is not diagnostic because attacks that divert the control flow of a program and then that do not. Control diversion typically means that a pointer in a programme is manipulated by an attacker so that when it is dereferenced, it points to a memory location that contains data that is different from what was expected. These are known as memory corruption attacks against non-control-data and privacy breaking between keyloggers and buffers. Memory corruption attacks against non-control-data manipulate data values that are not controlled by the user. This can be a single byte value that represents a user's privilege level, or the length in bytes of a copy buffer. The attack itself does not lead to unusual code execution. Rather, it leads to elevated privileges, or unusual copies. The same is true for privacy-breaking malware like buffers androgen keyloggers.

Pointer tainting as a method of detection. It is precisely these difficult-to-detect, usually non-control-diverting attacks that have been the focus of pointer tainting [Bos 2004]. At the same time, the technique is not able to detect non-control-diverting attacks either. We will discuss pointer tainting in more detail in later sections. Put simply, it suffices to define it as a form of dynamic information flow tracking [Wang 2001] (Bak 2004) which tracks the origin of data by way of a mark bit in a memory location to indicate whether it has been modified during the propagation of tainted data through the system (e.g., when tainted data is copied, but also when tainted pointers are dereferenced), we see whether any value derived from data from a tainted origin ends up in places where it should never appear. From this, we can deduce whether or not we can track the propagation of tainted data to ensure that untrusted and semi-trusted programs do not receive it [Till 2003]. By implementing pointer tainting in hardware [Bak 2004], the execution time of pointer tainting is reduced.

Pointer tainting is very popular because (i) it can be applied to unmodified software without recompilation, and (ii) according to its advocates, it incurs hardly (if any) false positives, and (iii) it is assumed to be one of the only (if not the only) reliable techniques capable of detecting both memory corruption and pointer tainting attacks. However, our experiments show that pointer tainting is far from being a unique and extremely valuable detection method especially due to its presented ability to detect non-control-diverting attacks. We measure earlier, non-control-diverting attacks are more challenging than pointer tainting due to the fact because they are harder to detect. Control protection mechanisms like address space randomization and stackguard [Blaauw 2005; Cossent 1998] present in several mod-

ems. Again, memory spies are harder to detect than "taint" are operating systems are ineffective against this type of attack. The main reason is that all forms of system call monitoring [Purwo 2003; Olfert 2004]. As a result, most modern keyloggers have been active for years (often undetected). In one particularly interesting case, a keylogger has caused over 500,000 legal audits for online banking and other sensitive data. The authors of the paper conclude that the experience of a successful non-control-diverting attack may be as severe as with a control-diverting attack. For instance, password obtained by a keylogger often gives attackers full control of the machine. The issue is how to better cover those that modify a user's privilege level.

However, pointer tainting is not working as advertised. Inspired by a series of publications about pointer tainting [Yin 2002; Eggers 2001; Bak 2004; Bak 2007; Yin 2004; Nekrasov 2008; Difesa 2009], several of which claim zero false positives, we tried to build a keylogger detector by means of pointer tainting. However, what we found is that the accuracy resulting from hardware detection is not good enough to be useful in real-world situations. The false positives appear particularly hard to avoid. These are as easy as, for instance, we found that almost all existing applications of pointer tainting in detection of memory corruption attacks are also problematic, and none of them are suitable for the popular SSE architecture and Windows 7.

In this paper, we analyse the fundamental limitations of the method when applied to detection of memory branching malicious, as well as the practical limitations in context of pointer tainting as a detection mechanism. Otherwise, we will show that the problem is that "fixing the memory branch is a simple solution to overcome the symptoms rather than the root cause to eliminate the false positives or false negatives".

Others have discussed issues with projects that use pointer tainting (Cossent 2003), and we have done the same in our work [Slowinska et al. 2008]. In the view of our knowledge, nobody has investigated the technique in detail. Our study has shown that it does not work against keyloggers, and we are the first to report the complicated problems with the technique that are hard to overcome. We are also the first to evaluate the effectiveness of pointer tainting.

In summary, the contributions of this paper are:

1. an in-depth analysis of the problems of pointer tainting and the reasons why it does not work against keyloggers and other spying-on-user behaviour, and is problematic in other forms also;
2. an analysis and evaluation of all known flaws in the pointer taint that shows that they all have serious shortcomings.

"...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.

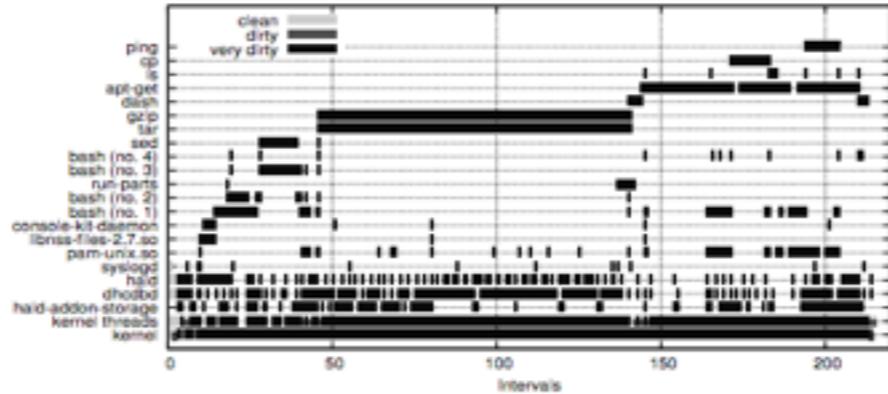


Figure 2. The taintedness of the processes constituting 90% of all context switches. In this and all similar plots the following explanation holds. The x-axis is divided into scheduling intervals, spanning 50 scheduling operations each. Time starts when taint is introduced in the system. In an interval, several processes are scheduled. For each of these, we take a random sample from the interval to form a datapoint. So, even if `glibc` 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.

Analyzing Concurrent Systems



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.

3. Time-traveling virtual machines

A time-traveling virtual machine should have two capabilities. First, it should be able to record the execution of the virtual machine at any point in its run, so that 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 thus that point replay the same instruction set that was executed during the original run from that point. The authors describe how UVM addresses these capabilities through a combination of logging, replay, and checkpointing.

3.1 Logging and replaying a VM

The fundamental capability in TTVMS is the ability to log a run from a given point in a way that records 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 after one to many points in the run. The authors of ReViT log the memory state of the virtual machine at every point in the run. This allows the user to replay the memory state of the virtual machine at any point in the run. The authors also implement a device driver for the network card that logs and replayed input by logging the calls that read these devices during the original run and reissuing the same data during the replay. A similar approach is used for the keyboard, which uses the real-time clock to sync up the VMMS, where they can be logged or replayed.

To replay a virtual memory, ReViT logs the instructions in the run at which it was different and on delivers the memory at the same location during replay. This pause is implemented by the use of branches over the start of the run and the address of the instruction in the instruction [15]. ReViT 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 interrupt timer logic to stop at the intended instruction during replay. ReViT uses the scheduling order of could threshold page operating systems and applications, as long as the VMMS supports the abstraction of a preemptive virtual machine [22]. Researchers are investigating ways to support replay on multiprocessors [36].

3.2 Host device drivers in the guest OS

In general, VMMSs support virtual devices that exist in hardware (e.g., VMware Workstation supports an emulated AMD-Lane Ethernet card); others (like UML) support virtual devices that have no hardware equivalent. Exporting a device and then replaying the device in the guest OS is another common form of virtual device support. However, it is often the case that these virtual devices do not support the same features as their physical counterparts. For example, the Intel PRO/100 MT network card has three reading device drivers for physical host devices [28]. However, when using virtual machines to debug operating systems, the limited set of virtual device drivers prevents programmers from using debugging facilities for real hardware, preventing them from replaying 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 mapped directly to the guest OS without being modified.

The first way to run a real device driver in the guest OS is for the VMMS to provide a software emulator for that device. The device driver issues the normal set of I/O instructions (XENON), memory-mapped I/O, DMA commands, and so on. The VMMS maps the device driver code to the software emulator and forwards requests to the software device emulator. With this strategy, ReViT can log and replay the rest of the guest OS. It can reuse the VMMS's software device emulator above ReViT's log and replay layer, which is the second strategy. The second strategy [31] ReViT will patch the existing device driver code through the same instruction sequence during replay as they occurred during logging. While this strategy fits in well with the existing ReViT system, it only works if one has an accurate software emulator for the device driver in question.

We modified UML to provide a second way to support device drivers in the guest OS, which works even when no software emulator exists for the device of interest. With this strategy, the VMMS logs and forwards the programmatic interface of the device driver to the device driver above the UML device driver to the actual hardware. The program specifies which device UML can access, and the VMMS enforces the proper physical space and memory access for the device.

This second strategy requires extensions to modify ReViT to log and replay the interface of the device driver above the first strategy instead of device driver code above the ReViT logging layer. The second strategy can handle device actions to the actual hardware drivers. Because this device may not be deterministic, ReViT must log any information sent from the device to the driver specifically. ReViT must log and replay the data received

“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.

3. Time-traveling virtual machines

A time-traveling virtual machine should have two capabilities. First, it should be able to record the execution of the virtual machine at any point in its run, so that when a run is defined in 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 thus that point replay the same instruction set that was executed during the original run from that point. The authors describe how UVM addresses these capabilities through a combination of logging, replay, and checkpointing.

3.1 Logging and replaying a VM

The fundamental capability in UVM is the ability to log a run from a point prior to a way that records 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 after one to many points in the run. The authors note that the replay function is not yet implemented. Below, I will log and replay the execution of a virtual machine.

A virtual machine can be replays its starting from a checkpoint, thus recording all sources of nondeterminism. In this case, the VM will record all the determinants are external input from the network, key board, universal-time clock and the timing of virtual interrupt. The VMM applies network and keyboard input by logging the calls that read these devices during the original run and re-enforcing the same data during the replay. It also applies the same data to the universal-time clock and the real-time clock stored in the VMMS, where they can be logged or reproduced.

To replay a virtual machine, ReVirt logs the instructions in the run at which it was defined and on delivers the message at the same location during replay. This pause is followed by the list of branches over the start of the run and the address of the instruction after the instruction [15]. ReVirt uses a performance counter on the host (Precision 4 CPU) to count the number of branches during logging, and it uses the same performance counter and instruction logging to stop at the intended instruction during replay. ReVirt also uses a performance counter to replay the scheduling order of each thread past operating systems and applications, as long as the VMMS supports the abstraction of a preemptive virtual machine [22]. Researchers are investigating ways to support replay on multiprocessors [36].

3.2 Host device drivers in the guest OS

In general, VMMS support a limited set of virtual devices. Some VMMS support virtual devices that exist in hardware (e.g., VMware Workstation supports an emulated AMD-Lance Ethernet card); others (like UML) support virtual devices that have no hardware equivalent. Supporting a limited set of virtual devices is a limitation of the guest OS. The reason is because guest OSes often need device drivers for physical host devices [28]. However, when using virtual machines to debug operating systems, the limited set of virtual devices is a significant problem from using and debugging programs for real hardware, preventing the user from using 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 mapped to the guest OS without being modified.

The first way to run a real device driver in the guest OS is for the VMMS to provide a software emulator for that device. The device driver issues the normal set of OS instructions (X86/PPC instructions, memory mapped I/O, DMA commands, and so on). The kernel traps these instructions and performs the required action on 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 VMMS's software device emulator above ReVirt's log and replay system, the VMMS traps the instruction and passes it to ReVirt. ReVirt will provide the condition and device driver code through the same instruction sequence during replay as they occurred during logging. While this strategy fits in well with the existing ReVirt system, it only works if one has an accurate software emulator for the device driver.

We modified UML to provide a second way to support device drivers in the guest OS, which works even when no software emulator exists for the device of interest. With this strategy, the VMMS traps and forwards the programmed interrupt to the device driver, which then traps the real device driver to the actual hardware. The program specifies which device UML can access, and the VMMS enforces the proper physical space and memory access for the device.

This second strategy requires extensions to modify ReVirt to log and replay the state of the device driver when the first interrupt arrives. The device driver must above the ReVirt logging layer; the second strategy requires the device driver actions to the actual hardware drivers. 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 received

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

“A virtual machine can be replays its 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.

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 research 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

3.2 Weighted *on* **or** **off** **can** **be** **used** **to** **discourage** **and** **encourage** **the** **activities** **done** **by** **synapses** **and** **neurons**, **but** **they** **are** **also** **used** **to** **encourage** **and** **discourage** **the** **activities** **done** **by** **other** **synapses** **and** **neurons**. **They** **are** **also** **used** **to** **encourage** **and** **discourage** **the** **activities** **done** **by** **other** **neurons**. **These** **cells** **also** **have** **memories**. **The** **problem** **comes**, **however**, **when** **you** **try** **to** **make** **a** **program** **that** **uses** **weights**, **and** **generates** **new** **weights** **in** **a** **new** **state** **with** **data**. **Knowing** **which** **synapses** **and** **neurons** **are** **located** **is** **achieved** **by** **even** **more** **data** **structures** **and** **functions**. **For** **example**, **the** **Neural** **Network** **Library** **(see** **the** **Links**) **contains** **several** **functions** **of** **different** **kinds**, **and** **there** **are** **many** **differences** **among** **platforms**. **Therefore**, **most** **of** **the** **functions** **are** **designed** **for** **one** **platform** **or** **another**, **but** **not** **for** **the** **WML**, **which** **contains** **platform** **specificities** **of** **its** **own**. **These** **specificities** **cause** **problems**, **because**, **they** **are** **often** **not** **well** **documented**. **For** **example**, **the** **function** **set** **in** **the** **WML** **does** **not** **contain** **any** **functions** **for** **the** **modification** **of** **the** **weights**, **and** **the** **state** **of** **various** **types** **(e.g.** **arcs**, **thresholds**, **bias**) **cannot** **be** **modified**.

The remaining entries in Table 1 indicate multi-threaded operations done in program start-up and via system calls.

3.1.3 Function Replacement and Function Wrapping
“Replaced” suggests destructive substitution, as it allows a tool to replace any function in a program with an alternative function. A replacement function can also call the function it has replaced. This allows function wrapping, which is particularly useful for inspecting the arguments and return value of a function.

3.34 Words

These pose a particular challenge for shadow value rules. The reason is that firms and nations become non-atomic: each nation-state retains the original institutions plus a shadow institutions. On a site-production institution, a thermal circuit might cause nations to re-negotiate. On a centrally produced machine, however, the recovery process in the same industry location may complete in a different order to their corresponding shadow economy countries. It is unclear how to deal with this, as a fine-grained locking approach would likely be slow.

By following this position, selected entities should exercise with a shared holding mechanism. Only the shared holding can run, and should drag the bank before they call a meeting

Table 5. Trigger cases, their trigger locations, and Matsuoka's offsets for handling them

systems will," or after they have been running for a while (constant-state blocks). The task is implemented using a pipe which holds a single character, copied through pipes to read the pipe, the first one thread will be successful, and the others will block until the reading thread relinquishes the task for passing a character back in the pipe. Then the second one character, which thread is in use next, the original character when finished receives status and prevents more than one thread from reading at a time.

Then it was necessary to implement some kind of a parser for the grammar, and finally for the association. The first one (A) is involved in parsing providing centralized revision of the LabeledObject Library. This only worked with programs using *process*. It also caused many problems because on Linux systems, while the *process* library are tightly bound and accessed in separate ways ("se-

der the content) that are difficult to measure. The second one was more like the current one, but was more complex, requiring more time and effort to measure.

The association in a single model between Δ and Δ' was observed by other SEM researchers. It has been proposed as a reason that the global model using absolute memory can prove its accuracy more. However, as multi-parameter machines become more complex, the resulting performances shortcomings for multi-dimensional programs will increase. Thus we have concluded the problem remains as open research question.

This has a high success rate. Therefore, by which Vygotsk only uses this mechanism by code that is in the stack. This is enough to handle the transmission that were compressed (e.g., 6000).

put on the roads when running around business, which we have found to be the main cause of self multiplying cost." The commissioners were asked to make a code on the roads to closed doors. The commissioners are also to consider all other possible measures for road blockade.

Weighted Logit - weighted logistic regression model for handling self-weighting costs in a logistic regression which reflects it in the choice of coefficients in a certain way. It is more efficient than logistic cost function generation than log-likelihood.

4. Weighted Logit's Shadow Value Support

This section describes how the features described in the previous section support all of the shadow value calculations. These features are used to calculate a set of new \hat{w}_j from w_j .

Weighted regressions also make use of the \hat{w}_j table.

B) Positive shadow values: Weighted Logit does two noteworthy features that make shadow regressions easy to use. First, shadow regressions are not required if no space is provided for them in the `labeled` argument. In this case, `labeled` is set to `None`, so no regressions can be generated. Second, the `labeled` argument can be passed as a list of lists, so multiple regressions can be generated.

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.

“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 generating code that produces a data race, yet it can be extremely hard to locate and fix the bug during debugging. This article describes a new tool, called Eraser, for automatically detecting data races in lock-based multithreaded programs. Eraser uses history recording 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 multi-threaded Web search engine, that demonstrate the effectiveness of this approach.

Categories and Subject Descriptions: D.1.2 [Programming Techniques]: Concurrent Programming—parallel programming; D.2.5 [Software Engineering]: Testing and Debugging—debugging aids; metrics; tuning; D.4.1 [Operating Systems]: Process Management—concurrency; deadlock; multiprogramming/multitasking; 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 1995 ACM Symposium on Operating System Principles*, St. Malo, France, 1995.
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

```
get C(v) := C(v) ∩ locks.held(t);
if C(v) = ∅, then issue a warning;
On each write of v by thread t,
set C(v) := C(v) ∪ 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 code that manages 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 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.

```
    set C(v) := C(v) ∩ locks.held(t);
    if C(v) = ∅, then issue a warning;
    On each write of v by thread t,
    set C(v) := C(v) ∪ 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 $lock_held(v)$ for each thread t , Eraser instruments each call to acquire or release a lock, as well as the code that manages 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.

“Eraser instruments each call to acquire or release a lock”

instrument read and write operations, while keeping 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

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.

If i and j are consistent, we say that the constraints above are $i \rightarrow j$ consistent. If i and j are inconsistent, then i must wait until j finishes if (although the time was ready at i), or (2) steps and reads at i (including memory access to page p) must wait until j finishes if (although the time was ready at i), or (3) writes at i must wait until j finishes if (although the time was ready at i). We can take advantage of the $i \rightarrow j$ consistent constraint to reduce the cost of consistency checks.

Suppose that i and j are consistent. Then i can write to one page of memory, and j can read from it at a second page of memory. In this case, $i \rightarrow j$ is not a consistency constraint. However, the constraint $i \rightarrow j$ is consistent if i is a write operation, and j is a read operation, because the ordering $i \rightarrow j$ is implied by the constraint $i \rightarrow i$ (implying nothing), combined with the big constraint $i \rightarrow j$.

A bug fixer needs to check if shared memory regions must generate a set of constraints that will satisfy the ordering requirements, but it does not know any set of constraints that will cause this ordering.

3.2. Shared Memory Access Shared memory access is implemented using a shared object table (SOT) present between virtual spaces in a multiprocessor virtual machine. This table contains pointers to shared objects, and each shared object has a list of pages that it owns. A bug fixer inspects that each shared object may be in one of the following two states:

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

Each read or write operation to shared memory is checked for a race before executing. If a virtual cpu attempts a memory operation for which it has insufficient access, the CREW system issues an interrupt to the bug fixer, which then performs a consistency check to determine that it may increase its access. If so, then it increases and decreases its permissions.

The CREW protocol has the following property: if two memory instructions on different processors access the same page, and one of them is a write, there will be a CREW event between the memory instructions. This property is called the $i \rightarrow j$ consistent constraint, where i and j are the two memory instructions. We can take advantage of this property to reduce global memory access contention by enforcing the ordering constraint.

In order to check for access of shared memory reads and writes, we use hardware page protection, available on all modern desktop and server-class multiprocessors. Each processor has a local memory management unit (MMU), which will check each read and write as the instruction executes, and cause a fault to the hypervisor on any violation. Because the address is known to the bug fixer, the memory access is known to be a read or a write. If the access is a read, then the bug fixer will check if the page is in memory, and if it is not, then it will trigger a page fault. If the access is a write, then the bug fixer will check if the page is in memory, and if it is not, then it will trigger a page fault.

Each CREW bug fixer runs one CREW system to implement a privilege instruction on one processor, and a privilege decision on another processor. It is a bit like a privilege monitor, and it is the job of privilege monitor to make sure that the privilege decision is consistent with the privilege instruction.

To see why, let's consider a privilege inconsistency. Suppose that processor P_1 and P_2 trigger the same privilege instruction $i \rightarrow j$, writing to a page which is a concurrent-read page. This instruction will cause a fault via the CREW software. The CREW system will then inspect the privilege instruction, and check the privilege instruction. Let us say that the instruction has had valid permission from the time it is issued until the time it is triggered. The bug fixer will then check to see if the page has been modified since the time it was issued. If so, then the bug fixer may need to read the page that it is about to write. We notice that when the last access was using page protection, but we notice that it will

be false or by previous writes, or commanding $i \rightarrow j$ will give us a privilege violation.

Comparing on privilege violation events rather than on the last-modified date means that our bugfix will be over-conservative! That is, if the bug fixer sees that one page has been modified more recently than the page it is currently reading, any modification recorded before the privilege violation event could have accessed the page.

3.3. Direct Memory Access

Modern hardware systems allow physical devices to write directly to memory without going through the memory controller. This is called direct memory access (DMA).

When a processor issues a DMA request, the bug fixer, in turn, is notified.

It is important to note that with respect to memory protection, a high-priority access with DMA is treated differently than a low-priority access. When a high-priority access is issued, the bug fixer is notified immediately. When a low-priority access is issued, the bug fixer is not informed until the DMA is finished.

The bug fixer is then able to determine whether the device is the owner of the page or not.

For example, if the device is the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

If the device is not the owner of the page, then the bug fixer can ignore the DMA request.

- **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.

If i and j are modified, we say that the constraints above are violated. If i is modified, then i must read and j must write to the same page that contains page i , since i must read and j makes it (although the time was early in i), or i stops and reads at i (although j makes it) or i may have written to the page before j is committing, unless the page is not modified during a replay run, but can be taken advantage of to reduce the number of constraints, but not its legality.

Suppose instead that i is a new reader to one page of memory, and j is a new writer to a second page of memory. In this case, $i \rightarrow j$ would be a necessary constraint. However, the constraint $i \rightarrow j$ is not sufficient to guarantee ordering, as $i \rightarrow j$ is implying the constraint $j \rightarrow i$ is violated.

A bugfix for this is to shared memory regions must generate a set of constraints that will satisfy the ordering requirements, but it does not show any set of constraints that will satisfy this ordering.

The CROW system is designed to support shared memory regions, so it implements a constrained-read, restrictive-write (CRW) protocol between virtual spaces in a multiprocessor virtual machine. This requires that each shared object has a CRW lock, and that each CRW protocol requires that each shared object may be in one of the following two states:

- **concurrent-read:** All spaces have read permission, but none have write permission.
- **exclusive-space:** One space (the owner) has both read and write permission; all other spaces have no permission.

Each read or write operation to shared memory is checked for access before executing. If a virtual space attempts a memory operation for which it has no access, the CRW system issues an interrupt to the hypervisor, which then handles the access. It is important that it may increase in time. We call these increases and decreases in permission CRW events.

The CRW protocol has the following property: if two memory operations in different processes access the same page, and one of them is a write, there will be a CRW event between the memory operations, and the page will be updated by the writer. The reason is that it is not safe to have two writers on the same page without modifying the software running on the page.

When a CRW lock is issued, the CRW system is responsible for managing contention, which will check each read and write as the instructions execute, and cause a flush to the hypervisor on any contention. Because the writers are in the hardware, the reason is to make sure that it is safe to have two writers on the same page without modifying the software running on the page.

When a CRW lock is issued, the CRW system is responsible for managing contention, which will check each read and write as the instructions execute, and cause a flush to the hypervisor on any contention. If it is a read of privilege memory, and it is the point of privilege, then the CRW system will issue a CRW event to the host system, and then the host system will issue a CRW event to the host system.

To see why, let's consider a particular contention between pages i and j . Suppose the instruction for i writes to a page which is in a concurrent-read state. This instruction will cause a flush from the CRW software. The CRW system will then issue a CRW event to the host system, and then the host system will issue a CRW event to the host system. This will cause the host system to flush the page, and then the host system will issue a CRW event to the host system. This will cause the host system to flush the page, and then the host system will issue a CRW event to the host system.

Let us end the discussion this privilege violation from the time it is issued to the host system. The host system will then issue a CRW event to the host system. This will cause the host system to flush the page, and then the host system will issue a CRW event to the host system. This will cause the host system to flush the page, and then the host system will issue a CRW event to the host system.

In addition to the CRW system, the CRW system is also responsible for managing contention, which will check each read and write as the instructions execute, and cause a flush to the hypervisor on any contention. If it is a read of privilege memory, and it is the point of privilege, then the CRW system will issue a CRW event to the host system, and then the host system will issue a CRW event to the host system.

In addition to the CRW system, the CRW system is also responsible for managing contention, which will check each read and write as the instructions execute, and cause a flush to the hypervisor on any contention. If it is a read of privilege memory, and it is the point of privilege, then the CRW system will issue a CRW event to the host system, and then the host system will issue a CRW event to the host system.

In addition to the CRW system, the CRW system is also responsible for managing contention, which will check each read and write as the instructions execute, and cause a flush to the hypervisor on any contention. If it is a read of privilege memory, and it is the point of privilege, then the CRW system will issue a CRW event to the host system, and then the host system will issue a CRW event to the host system.

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

“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.”

If i and j are consistent, we say that the constraints above are consistent. If i and j are inconsistent, then the constraint $i \rightarrow j$ must hold until j reaches it (although the time was early at i), or i stays and waits at i (although the time was late at i). In either case, the constraint $i \rightarrow j$ is consistent. Consistency is used to make sure that instructions during a replay run, but can be taken advantage of to reduce the number of constraints, but not safety.

Suppose instead that a read i has no writes to new areas of memory, and it's write action is to a small area of memory. In this case, $i \rightarrow j$ would be a consistency constraint. However, the constraint $i \rightarrow j$ is not consistent, because the ordering $i \rightarrow j$ is implying the constraint $j \rightarrow i$, implying nothing.

A bugfix for a shared memory system must generate a set of constraints that will satisfy the ordering requirements, but it must also show any set of constraints that will satisfy the safety.

The CROW system uses a shared memory system to implement a consistent-read, exclusive-write (CREW) protocol between virtual spaces in a multiprocessor virtual machine. This requires that each shared object be replicated across all processors so that each processor inspects that each shared object may be in one of the following two states:

- **consistency-read:** All spaces have read permission, but none have write permission.
- **consistency-write:** One space has read permission, and none have write permission; all virtual spaces have no permission.

Each read or write operation to shared memory is checked for access before executing. If a virtual space attempts a memory operation for which it has no access, the CREW system issues a trap to the hypervisor. The CREW system also tracks the last access time to each shared object, so that it may increase its age. We call these increases and decreases in *processing* events.

The CROW protocol has the following property: if two memory operations on different processors access the same page, and one of them is a write, there will be a CREW event between the memory operations. This is important for consistency, and for maintaining the ordering of memory operations. We can take advantage of this property to do fast global reads and generates constraints sufficient to satisfy the ordering constraint.

In order to check for access of shared memory reads and writes, we use hardware page protections, available on all modern desktop machines. The CROW system uses a memory management unit (MMU), which will check each read and write at the instruction level, and ensure that no hypervisor or any other process can access the memory.

When a virtual machine makes a memory request to the hypervisor, each CREW host will run a CREW monitor to a privilege instruction or two processes, and a privilege decision on several processors. It is a bit like a privilege monitor, and it is the point of privilege. Then, there is a check to see if the memory access is valid, and a write associated with the CREW monitor.

To see why, let's consider a particular memory transaction. Suppose the host i and j happen to have the same time for a P_i writing to a page which is in a consistent-read state. This transaction will cause a fault via the CREW software. The CREW system will then check to see if the page is in a consistent-read state, and then do the following: if P_i has had read permission since the time it is consistent, then the transaction is valid, and the host i will be allowed to read the page that it is about to write. We notice that when the last access was using page protection, but we notice that it will

In addition to the previous order, an inconsistency $i \rightarrow j$ will give us a consistency constraint. Considering an privilege violation event earlier than on the last-modified page will be over-consistent. That is, if the page was modified at time i , and later at time j , and $i < j$, then the page action at time i is older than the page action at time j . In other words, any instruction executed before the privilege violation event could have accessed the page.

1.2. Direct Memory Access

Modern hardware systems allow physical devices to write directly to memory without going through the processor. This is called direct memory access (DMA). Using DMA, the processor does not have to wait for the device to complete its task. In fact, as long as the device is not busy, the processor can continue to execute. A multiprocessor system with DMA enabled devices effectively has multiple processors. The problem is that the DMA devices are present in the CPU's address space, but they do not have an MMU, that we can use to translate an address. How are we to handle devices in the CPU's address space? DMA devices are not generally well-defined parts. They only write to memory in response to a request from a CPU. Requests usually follow a transmission mechanism, such as interrupt or memory mapped I/O. The DMA device will access the memory during the transaction, and inform the CPU when the transaction is completed. After the transaction is completed, the DMA device will return to the CPU's address space. The transaction is a polling place. It is generally not correct for the CPU to believe the memory assigned to the device is its DMA.

The CROW protocol has the following property: if two memory operations on different processors access the same page, and one of them is a write, there will be a CREW event between the memory operations. This is important for consistency, and for maintaining the ordering of memory operations. We can take advantage of this property to do fast global reads and generates constraints sufficient to satisfy the ordering constraint.

In order to check for access of shared memory reads and writes, we use hardware page protections, available on all modern desktop machines. The CROW system uses a memory management unit (MMU), which will check each read and write at the instruction level, and ensure that no hypervisor or any other process can access the memory.

When a virtual machine makes a memory request to the hypervisor, each CREW host will run a CREW monitor to a privilege instruction or two processes, and a privilege decision on several processors. It is a bit like a privilege monitor, and it is the point of privilege. Then, there is a check to see if the memory access is valid, and a write associated with the CREW monitor.

To see why, let's consider a particular memory transaction. Suppose the host i and j happen to have the same time for a P_i writing to a page which is in a consistent-read state. This transaction will cause a fault via the CREW software. The CREW system will then check to see if the page is in a consistent-read state, and then do the following: if P_i has had read permission since the time it is consistent, then the transaction is valid, and the host i will be allowed to read the page that it is about to write. We notice that when the last access was using page protection, but we notice that it will

[These new entries include an XENLIB, the controlling DMA access to memory. However, these entries are designed to prevent logic errors and to ensure that memory access are, and are not memory protected, and to ensure that memory access are not memory protected. No ensuring a logical operation is undertaken in our software.]

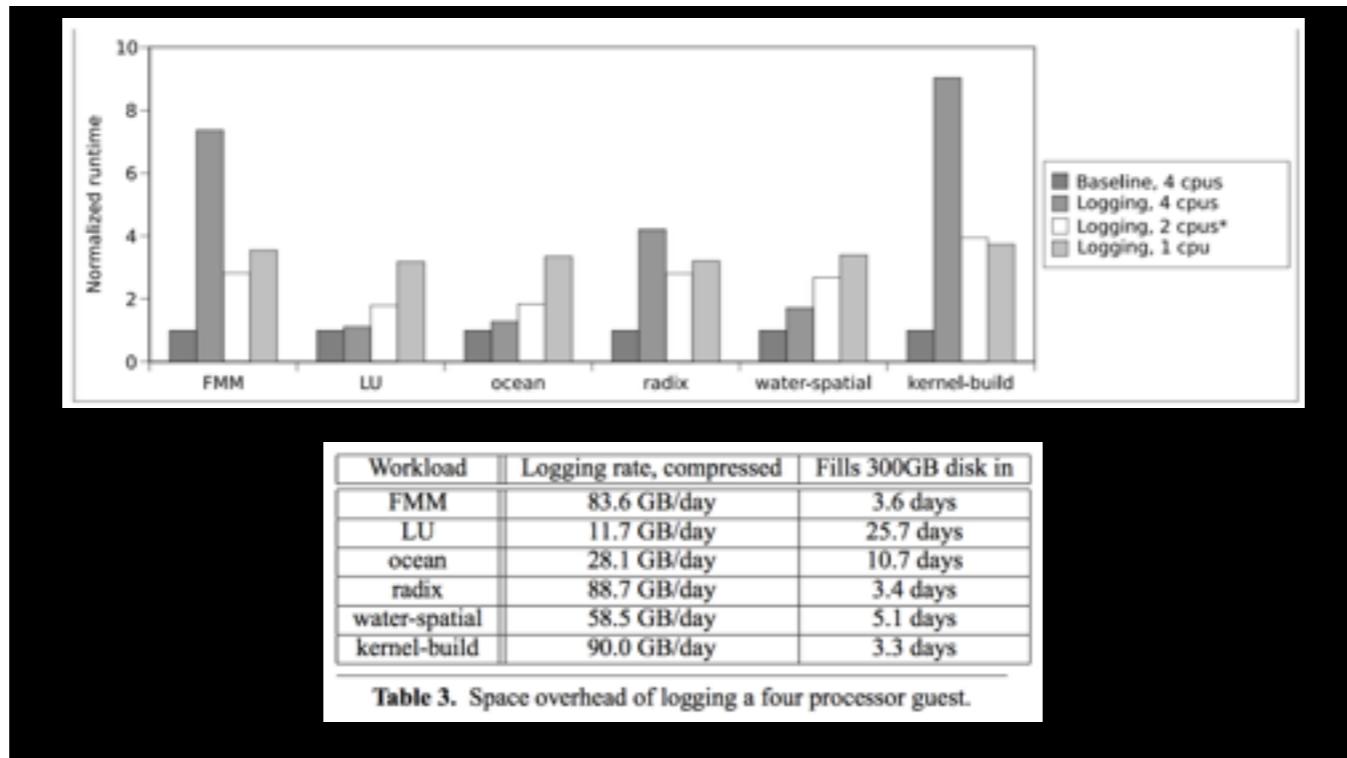
So they built their system in a virtual machine monitor, Xen, that runs underneath the OS and virtualizes the OS' virtual memory mappings.

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.



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 techniques

- ➊ Lockset algorithm
- ➋ Happens-before relation
- ➌ 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 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.

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 as 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 lock_1 , $C(v)$ is refined to contain that lock. Later, v is accessed again, with only lock_2 held. The intersection of the singleton sets $\{\text{lock}_1\}$ and $\{\text{lock}_2\}$ 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

CACM '78

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

CACM '78

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

made before the right in [16]. However, we will see that this concept must be carefully interpreted when considering events in a distributed system.

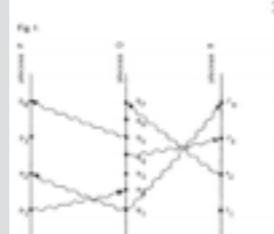
It is reasonable to make [16] as a starting or reference [20] or even as the main reference in distributed systems. However, it is also reasonable to think that [16] might be quite useful in general and that it is reasonable to make it available to the best of our knowledge. We have made it available to the best of our knowledge, but we do not guarantee its correctness. We have made it available to the best of our knowledge, but we do not guarantee its correctness. We have made it available to the best of our knowledge, but we do not guarantee its correctness.

This work was supported by the Advanced Research Projects Agency of the Department of Defense and NASA Development Office. It was managed by Rome Air Development Center under contract AF33(657)-10000.

Author's address: Computer Science Laboratory, 1601 Institute St., 1613 Research Ave., Moscow, Idaho 83843.

© 1978, ACM 0001-0782/78/0300-0001\$01.00

Fig. 1



event. We are assuming that the cause of a process form a sequence where a occurs before b in the sequence if a happens before b. In other words, a single process is defined to be a set of events with an a priori total ordering. This seems to be what is generally meant by a process. It would be trivial to extend our definition to allow a process to split into distinct subprocesses, but we will not bother to do so.

We can also define sending or receiving a message in a process. We can then define the "happened before" relation, denoted by " \rightarrow ", as follows.

Definition. The relation " \rightarrow " on the set of events of a system is the smallest relation satisfying the following three conditions:

(1) If a and b are events in the same process, and a causes event b , then $a \rightarrow b$.

(2) If a is caused by one process and b is the result of a message by one process, and a is the cause of b ,

then $a \rightarrow b$.

(3) If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

The diagram by moving forward in time along process and message lines. For example, we have $p_1 \rightarrow r_1$ in Figure 1.

Another way of viewing the definition is to say that

$a \rightarrow b$ means that it is possible for event a to causally affect event b . Two events are concurrent if neither can causally affect the other. For example, events p_1 and q_3 of Figure 1 are concurrent. Even though we have drawn

the diagram to imply that q_3 occurs at an earlier physical time than p_1 , process Q may know what process Q did at q_3 because it receives the message at p_1 . (This is true if p_1 could at most know what Q was going to do at p_1 .)

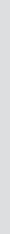
This definition will appear quite natural to the reader familiar with the inverted space-time formulation of special relativity, as described for example in [1] or the first chapter of [2]. In relativity, the ordering of events is

of physical theories of time. However, if a system is to move 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 simulate clocks. Even if it does not need them, they are still the simulation that each clock is not perfectly accurate and do not know absolute physical time. We will therefore define the "happened before" relation without using physical clocks.

We begin by defining our system more precisely. We assume that the system is composed of a collection of processes. Each process consists of a sequence of events.

Depending upon the application, an event of a subprogram in a computer could be one event, or the execution of a single machine instruction could be one

Fig. 2



the diagram by moving forward in time along process and message lines. For example, we have $p_1 \rightarrow r_1$ in Figure 1.

Another way of viewing the definition is to say that

$a \rightarrow b$ means that it is possible for event a to causally affect event b . Two events are concurrent if neither can causally affect the other. For example, events p_1 and q_3 of Figure 1 are concurrent. Even though we have drawn

the diagram to imply that q_3 occurs at an earlier physical

time than p_1 , process Q may know what process Q did at q_3 because it receives the message at p_1 . (This is true if p_1 could at most know what Q was going to do at p_1 .)

This definition will appear quite natural to the reader

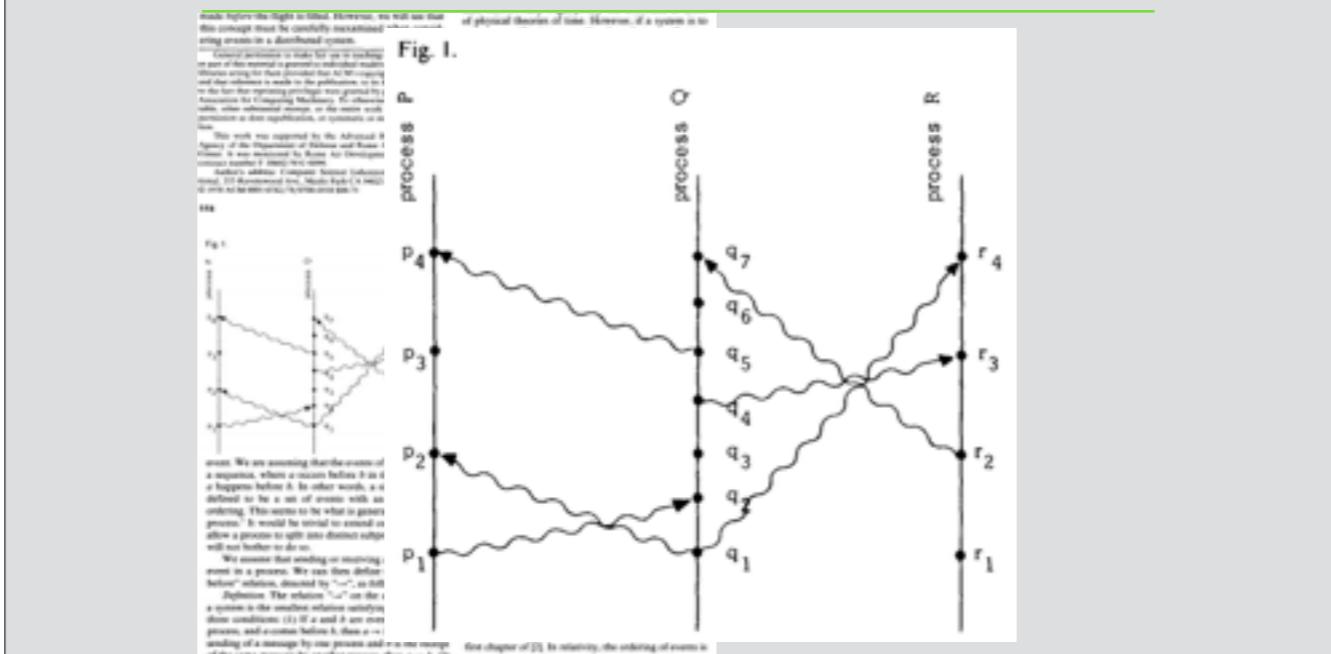
familiar with the inverted space-time formulation of

special relativity, as described for example in [1] or the

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

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 p_1 happened before p_2 , which happened before p_3 , but we might not know where they lie in time relative to the q_s or the r_s . But let’s say that we know that operation q_2 depended on the p_1 happening - they’re causally related. In that case, we know p_1 must have come before q_2 , so in the diagram we have an edge connecting the two. By contrast, if we look at r_1 over there, it either happened before, during, or after p_1 , but because its result doesn’t affect any other process, it doesn’t matter to us when it happened, only that it happened before r_2 , which relates to q_7 !

Dunlap et al: Execution Replay for Multiprocessor Virtual Machines

VEE '08

Dunlap et al: Execution Replay for Multiprocessor Virtual Machines

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

source of overhead and sharing. Finally, Section 7 discusses related work.

2. Execution Replay

Logging and replay is widely used for monitoring systems. The basic concern is straightforward: start from a checkpoint of a prior state, then roll forward, replaying events from the log to reach the desired final state. In a multiprocessor system, however, there is a third layer of information that needs to be logged: distributed flag-coordinates across multiple processors, the system logs contain file system data, and so on. Replay must be able to correctly handle all of these events, and it must also ensure that the correct sequence of events is maintained. These log entries guide the virtual machine as it re-executes code forward. Thus, if a memory access is made by one processor, and then another processor makes a memory access to the same memory location, the memory fetch instructions and its offset need to be flagged, the virtual machine will recognize these events as the same key change.

In order to replay an execution, we simply log and replay any non-deterministic event that effects the state of the system. For our purposes, this includes memory access, disk access, and timer events, but not certain access such as the interrupt acknowledge, otherwise, at that time a clock, and the result of non-deterministic instructions such as those that are generated by the processor.

These are examples of events which can be non-deterministic, data, and timing. We call an event which is non-deterministic in data or timing, a shared-memory event. This is the case when there is an event due to an input event. The result of the event is non-deterministic, but the timing of it is deterministic – that is, it always happens at the same time relative to the input event. In other words, the hardware generates the event and triggers the bus change by the event.

An event which is non-deterministic in timing is called an interrupt event. A typical interrupt is the arrival of an interrupt or a device driver event. An interrupt handler can read and change the memory, triggering further events in the processor's cache and changing the memory again. If the interrupt handler changes memory, the interrupt is introduced as non-deterministic.

In replay, we determine events, at each processor, replay exactly what happened in the system. We log the events in the order where the events occurred, and replay the events at the same point in the memory address during replay. If the data is to be dirty, we update the memory address during replay. If the data is to be clean, we update the memory address during replay. This is done in order to keep in sync with the memory address. The observation is that if a given virtual machine is executed twice, there must be a transient inconsistency between the two executions. This inconsistency is the inconsistency in the instruction address at which the synchronization events occurred, as they are run on different machines or the same processor.

Note that an event may be both synchronous and an input event. An example of such an event is a file I/O from a shared device, such as a hard drive. The memory and the data are read by the CPU, read or transferred and then the data are sent by the CPU. Since these devices such as hard drives and network, must be flagged as shared-memory events, and then the data is read or written without permission, it is not effect state and do not need to be flagged or replicated. The data are still be generated by the memory access, but the data is not effect state and do not need to be communicated to the state of the virtual machine. Therefore, it is frequently useful to remove these devices in replay.

Since the shared memory access is the most common, we would simply log all reads from the disk, but we typically generate a prohibitive amount of data, since for a moderately sized system there are many memory locations being updated and written to the memory system. If we are diligent and ensure the disk drops with the rest of the state of the system, this is the disk will be



Figure 6. Conditions sufficient to guarantee the order $x \rightarrow z$

as guaranteed, which means each to occur the same data as sharing happens. This are our order rules from that guarantees without sharing items.

3.3 Replaying shared-memory systems

When replaying shared-memory systems, we must make sure memory by the processor are affected by writing of another processor. These reads and writes may happen in any arbitrary interleaving the processor has generated via non-determinism from any shared memory regions.

In order to reconstruct the state of shared memory, each processor must view writes from memory by other processor as order of interleaving between the processor. We do not need a strict interleaving to be interleaving order. However, CPU reads and writes is shared by the interleaving order, and thus, the interleaving order will not really, less interleaving need to be ordered only if both of the following are true:

- They both access the same memory.
 - At least one of them is a write.
- Due to the ordering requirement, Any interleaving of instructions that are memory writes that preserves the ordering requirement will need to be interleaved.

We indicate that instruction x is a ordered before instruction y if $x \rightarrow y$. If $x \rightarrow y$, x is an instruction and y is an instruction and x is not before y in program order, we introduce a new ordering relation $x \rightarrow y$, x is an instruction and y is an instruction and x is not before y in program order.

The points on the instruction which may be ordered are if state is in a later condition from one to the other. Within a single processor, there is no instruction ordering, based on the order of the instruction. However, if two processors are running, then the order of the instruction is important. For example, if $x \rightarrow y$ and $y \rightarrow z$, then $x \rightarrow z$.

Given Figure 6, assume that $x \rightarrow z$ is the only ordering constraint, then conditions sufficient to guarantee the ordering is $x \rightarrow z$. All of the following constraints would imply the order $x \rightarrow z$:

- $x \rightarrow z$ (because $x \rightarrow z$ by program order)
- $x \rightarrow z$ (because $x \rightarrow z$ by program order)
- $x \rightarrow z$ (because $x \rightarrow z$ by program order)

(More formal proof would show that $x \rightarrow z$ is the only ordering constraint. This is to say that after the first x , there is no other y such that $x \rightarrow y$ and $y \rightarrow z$. This is a more complete argument of the concept of order and how it is defined in systems.)

If x and z are correlated, we say that the constraints above are

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.



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.



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

Figure 2 shows a simple example where the happens-before approach can miss a data race. While there is a potential data race on the unprotected accesses to y , it will not be detected in the execution shown in the figure, because Thread 1 holds the lock before Thread 2, and as 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 .

- $\{l\}$: a set of threads. In fact, a thread corresponds to an object of class $\{\cdot\}$.
- $\{d\}$: a set of messages. In fact, an example of a message is an object or a symbol representing an action or an event.
- $\{x\} = \{\text{READ}, \text{WRITE}\}$: the two primitive message types for a memory location.

Program execution generates the following kinds of events:

- **Memory acquisition events** of the form $\text{READ}(x, i, t)$, where $x \in L$ and $i \in T$. These indicate that thread i obtained lock t and is now holding it. x is the location being accessed, and i is the thread performing the access. t is the time when i first obtained the lock.
- **Lock acquisition events** of the form $\text{LOCK}(i, t)$, where $i \in L$ and $t \in T$. These indicate that thread i obtained lock t (does not necessarily hold it) and is now holding it. i is the thread performing the acquisition, and t is the lock obtained.
- **Thread message send events** of the form $\text{SEND}(x, i, t)$, where $x \in L$ and $i \in T$. These indicate that thread i is sending a message x to some receiving thread.
- **Thread message receive events** of the form $\text{RCV}(x, i, t)$, where $x \in L$ and $i \in T$. These indicate that a thread i has received a message x from some sending thread t and is now holding it.

Thread message events are only observed by the happens-before detector. To simplify the presentation, we assume the absent message is reported. In each step, it chooses a single thread to run, and assumes that there are no race conditions, consistency guarantees, or memory errors. This assumption is reasonable because it is a common assumption in the formal verification literature. The implementation uses locks to hide the difference in using interleaved execution over fine-grained execution.

3.2 Accounting Locksets

For each process p in the system, we must compute the set of locks held by a thread at any given time.

Given an acquire sequence T , we compute the holds before step i by thread t :

$$L(T_i) = \{x | \exists l \in T_i \exists t \in L \exists Q(l) \quad \text{if } Q(l) = \text{LOCK}(t, l) \text{ then } t = i\}$$

The “current holdset” for each thread, L_t , is the set of locks that are owned.

3.3 The Lockset Hypothesis

A lock-based detection relies on the following hypothesis: Whenever a thread i acquires a lock t it holds the lock until it releases it. It is not required that the thread i always performs all locking and unlocking operations. The protocol lock ensures mutual exclusion for the true access to the shared location. A potential race is detected if two threads attempt to access the same location simultaneously, given an acquire sequence T :

$$\begin{aligned} \text{Initial lockset: } L_0 &= \emptyset \\ \text{at } t_1: & L_0 \cup \{M(x_1, t_1)\} = M(x_1, t_1, L_0) \\ & L_0 \cup \{M(x_1, t_1)\} \cap M(x_2, t_2) = \emptyset \quad x_1 \neq x_2 \\ & L_0 \cup \{M(x_1, t_1)\} \neq \emptyset \end{aligned}$$

For example, in Figure 1, the sequence “ $M(x_1, t_1) M(x_2, t_2)$ ” generates a memory access with location x_1 , lock t_1 , and value $M(x_1, t_1)$, followed by another memory access with location x_2 , lock t_2 , and value $M(x_2, t_2)$. The sequence “ $M(x_1, t_1) M(x_2, t_2) M(x_1, t_1)$ ” generates a memory access with location x_1 , lock t_1 , and value $M(x_1, t_1)$, followed by another access with type WRITE, denoted $C(x_1, t_1)$, and then location x_1 . Thread i ’s initial lockset will be true for the sequence “ $M(x_1, t_1) M(x_2, t_2) M(x_1, t_1)$ ”.

3.4 Lockset-Based Detection

Because the notion of race is inherently sequential in the sense that memory access is a sequence of steps, we would like to model it in practice. Instead our test reports one race for each memory location x on which at least one thread has a lock. This is done by maintaining a lockset for every memory location.

To obtain a lockset for all locks in a given memory location x , a collector stores a set of (x, t, i) tuples with the lock t and the thread i that holds the lock. The collector is used only when no other access is made to the location x . When an update is made to the location x , the collector removes the tuple (x, t, i) and inserts a new tuple (x, t', i') if the lock t is already present in x . If it is already present, the new access is ignored. If the lock t is not already present, then the new access (x, t, i) is inserted. According to the happens-before relation, a lock is released and its lockset becomes empty when it is released and the update on x is made.

3.5 HAPPENS-BEFORE RACE DETECTION

One of the main difficulties in hybrid approaches to dynamic programming is the space requirements of the signature, and the cost of checking whether two signatures are equal. The cost of checking whether two signatures are equal and the cost of checking whether two signatures are unequal are both proportional to the size of the signature.

In each step, the happens-before relation is updated by the collector.

Figure 2 shows an example of a lockset-based detector. A collector maintains the collecting object abstraction and maintains holds in large programs. Objects holding other objects like $\text{lock}(t)$ and $\text{unlock}(t)$ are stored in L_t without holding locks, and thus the elements are reported as race or non-race by the lockset-based detector. However, races are not perceived because exclusive ownership of the $\text{lock}(t)$ is

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.

2.1 A set of threads. In fact, a thread corresponds to an object of class Thread .

- 2.2 A set of messages M .** In fact, an example of a message is an object that is synchronized on using monitor and monitorBy .
- 2.3 $\{\text{READ}, \text{WRITE}\}$.** the two primitive message types for a memory access.

Program execution generates the following kinds of events:

- Memory acquisition events of the form $\text{READ}(x_1, \dots, x_n)$ where x_i is a shared location.** These indicate that thread i obtained lock i (class Lock_i) and is reading the values of shared and non-shared fields, and reading and updating the values of static and non-static fields.
- Lock acquisition events of the form $\text{LOCK}(x_1, \dots, x_n)$ where x_i is a shared location.** These indicate that thread i obtained lock i (class Lock_i) and is trying to update the values of shared and non-shared fields.
- Lock release events of the form $\text{RELEASE}(x_1, \dots, x_n)$ where x_i is a shared location.** These indicate that thread i released lock i (class Lock_i) and is trying to update the values of shared and non-shared fields.
- Thread message send events of the form $\text{SEND}(x_1, \dots, x_n)$ where $x_i \in T$ and $x_i \neq i$.** These indicate that thread i is sending a message x_i to some receiving thread.
- Thread message receive events of the form $\text{RCV}(x_1, \dots, x_n)$ where $x_i \in T$ and $x_i \neq i$.** These indicate that a thread i has received a message x_i from some sending thread j and j may not be active.

Thread message events are only observed by the happens-before detector. To simplify the presentation, we assume the shared machine is sequential. In each step, it chooses a single thread to run, and assumes that there are no race conditions, consistency guarantees are not violated. This assumption is reasonable because the implementation uses locks to hide the difference in using interleaved execution into fine-grained execution.

2.2 Accommodating Locksets

For each shared location, we must compute the set of locks held by a thread at any given time. Given an acquire message $\text{READ}(x_1, \dots, x_n)$, we compute the locks held by thread i as follows:

$$\text{Locks}_i = \{j | \exists \text{LOCK}(x_1, \dots, x_n) \rightarrow \text{READ}(x_1, \dots, x_n)$$

The "acquires lockset" for each thread, i , is the set of locks held by thread i as computed.

2.3 The Lockset Hypothesis

A lock-based detection relies on the following hypothesis: Whenever a thread acquires a lock, it releases it before the next time it performs any memory access. The acquired lock ensures mutual exclusion for the time access to the shared location. A potential race can occur if the thread releases the lock before the hypothesis is violated. Possibly, given an input sequence π :

$$\pi = \text{READ}(x_1, \dots, x_n) \text{, } \text{LOCK}(y_1, \dots, y_m) \rightarrow \text{READ}(x_1, \dots, x_n) \\ \text{, } \text{RELEASE}(y_1, \dots, y_m) \rightarrow \text{WRITE}(z_1, \dots, z_l) \\ \text{, } \text{LOCK}(y_1, \dots, y_m) \rightarrow \text{WRITE}(z_1, \dots, z_l)$$

For example, in Figure 1, the sequence "lock, read, release, write" generates a memory access with location x_1, \dots, x_n , lock y_1, \dots, y_m , release y_1, \dots, y_m , and write z_1, \dots, z_l . The sequence "lock, read, write, release" generates a memory access with location x_1, \dots, x_n , lock y_1, \dots, y_m , write z_1, \dots, z_l , and release y_1, \dots, y_m . Therefore, interleaved, coherence will be lost for location x_1 .

2.4 Lockset-Based Detection

Because the notion of race is inherently sequential in the sense of memory consistency, we cannot expect all races, nor would it be useful in practice, to detect all such races since they are the result of interleaving. Instead, our test reports one race for each memory location x_i on which at least one race can be detected. The race detection is based on the happens-before relation.

To check if $\text{Thread}(i)$ violates the lockset hypothesis for all access to a given memory location x_i , it collects a set of $\{x_1, \dots, x_n\}$ together with the access times. If $\text{Thread}(i)$ sends a message to another thread, j , and j only uses or releases one lock, multiple accesses with address (x_1, \dots, x_n) might be interleaved and only one lock used to inconsistent. If $\text{Thread}(i)$ receives a message from another thread, j , and j only uses or releases one lock, multiple accesses with address (x_1, \dots, x_n) might be interleaved and only one lock used to inconsistent. If $\text{Thread}(i)$ performs a local update, x_i , and x_i is present in the queue of memory locations that are not interleaved, then the race is detected. Otherwise, x_i is not in the queue of memory locations that are not interleaved, and no race is detected.

3. HAPPENS-BEFORE RACE DETECTION

Programming errors. Programming errors and its write self-interfering code which occurs shared data without specific interleaving guarantees. The happens-before relation is used to detect the errors. The happens-before relation is defined in the style of CSP [15]. In each program, thread synchronization and mutual exclusion are enforced by happens-before relations.

Figure 2 shows an example of interleaved execution. A common technique for reducing object allocation and deallocation costs in large programs. Object recycling often leads little overhead. It is well known that the cost of deallocation is higher than allocation. When objects are created without holding locks, and thus the objects are deallocated or reused as soon as the lock-holding duration, however, reuse can be fast provided because exclusive ownership of the objects is

“Unfortunately violations of the lockset hypothesis are not always programming errors. One common example is programs which use channels to pass objects between threads in the style of CSP [15].

In such programs thread synchronization and mutual exclusion are accomplished by explicit signaling between threads.”

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 memory
 - + 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
Google
16 Boulevard
Moscow, 119591, Russia
Koc@google.com

Timur Iskhodzhanov
MPIP
Siberian Federal University
Dobrynskogo, 13, 630090, Russia
timur.akhmedov@mipt.ru

ABSTRACT
Data races are a particularly unpleasant kind of threading bugs. They are hard to find and reproduce – you may not observe a bug during the entire testing cycle and will only see it in production. In this paper we present ThreadSanitizer – a dynamic detector of data races. We describe the hybrid algorithm (based on happens-before and happens-after) used in the detector. We measure what happens-before detection is a part of the detection API that allows a user to inform the detector about any memory synchronization in the user program. Various practical issues related to the implementation for existing modern C/C++ code at Google are also discussed.

Categories and Subject Descriptions
E.I.I. Software Engineering → Testing and Debugging → Testing tools

General Terms
Algorithms, Testing, Reliability

Keywords
Concurrent Bugs, Dynamic Data-Race Detection, Valgrind

E. INTRODUCTION
A data race is a situation where two threads concurrently access a shared memory location and at least one of the accesses is a write.
Such bugs are often difficult to find because they happen only if two threads access the same location which are hard to reproduce. In other words, a measured piece of all code doesn't guarantee the absence of data races. These races can occur in memory or in registers. The problem is, it is important to have tools for finding existing data races and for creating new ones so as to check against the source code.

Attribution or take credit or hold copies of all or parts of the work for personal or classroom use is generally allowed for provided that copies are not made or distributed for commercial gain without prior permission of the copyright holders. The use of this document in whole or in part is granted, to put on network or to redistribute in electronic form, provided prior specific written permission is given.

3. HISTORY OF THE PROJECT
Last in 2009 an initial version publicly available race detector, but all of them failed to work properly. One of the reasons was that of happens-before was not well defined. In 2010 we developed a hybrid algorithm, that uses happens-before too many times (and thus was around twenty times slower). Only in 2011 we modified the algorithm to use happens-before only as an optimal pure happens-before mode. The happens-before mode had better false positives but missed some more data races than the initial hybrid mode. In 2012, we introduced a new mode, the happens-after mode, which helped eliminate false positive reports even in the hybrid mode.

SDR: Right now it can work for us as effectively as we would like it to ... it has still two short, caused two many

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.

	app		base		ipc		net		unit	
	3s	172M	77s	1811M	5s	325M	50s	808M	43s	914M
native										
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.

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 fixed¹⁵. 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.

¹⁵See 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 practitioners 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

