Execution Mining

Geoffrey Lefebvre Brendan Cully Christopher Head Mark Spear
Norm Hutchinson Mike Feeley Andrew Warfield
Department of Computer Science, University of British Columbia
{geoffrey, brendan, chead, mspear, norm, feeley, andy}@cs.ubc.ca

Abstract

Operating systems represent large pieces of complex software that are carefully tested and broadly deployed. Despite this, developers frequently have little more than their source code to understand how they behave. This static representation of a system results in limited insight into execution dynamics, such as what code is important, how data flows through a system, or how threads interact with one another. We describe Tralfamadore, a system that preserves complete traces of machine execution as an artifact that can be queried and analyzed with a library of simple, reusable operators, making it easy to develop and run new dynamic analyses. We demonstrate the benefits of this approach with several example applications, including a novel unified source and execution browser.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Tracing, Diagnostics

General Terms Design, Experimentation, Measurement, Performance

Keywords Binary Analysis, Offline Analysis, Virtual Machine, Semantic Gap

1. Introduction

> *"The creatures were friendly, and they could see in four dimensions. They pitied Earthlings for being able to see only three. They had many wonderful things to teach Earthlings about time."*
>
> –A description of the Tralfamadorians from Kurt Vonnegut's "Slaughterhouse Five" [35]

Execution is fleeting. As each moment of execution is lost to the past, an important opportunity to understand a system's behavior may be lost as well. In addition to the complexity that is inherent in operating system code, developers are often hamstrung by a lack of useful tools to assist in understanding how source behaves in practice. Where tools such as debuggers, profilers and leak detectors are available at all, they operate on only a glimpse of execution, either as a point in time or as a predetermined summary. These tools are "stuck in time", and must anticipate or recreate any execution scenario they seek to examine. Execution behavior that is not anticipated before it occurs can not be examined.

The lack of insight into how an operating system actually behaves when it runs presents a major challenge to both novice and experienced developers: code bases for these systems are large, and control flow through a system – whether it has to do with servicing a system call or delivering a received packet to the appropriate application – frequently spans large numbers of source files and is indirected through function pointers and across multiple execution contexts. A view of program source alone does not provide an intuition for where common execution paths are, which regions of code are particularly performance sensitive, or how data structures such as packets or block requests move across subsystems.

In this paper, we explore the idea that detailed, CPU-level traces of system execution can usefully enable analysis on execution as a whole, as opposed to execution in the moment. Rather than considering dynamic analysis as a "one-off" task that is performed against a running system, we propose performing repeated analysis against a persistent trace instance that may be kept and analyzed over a long period of time and by multiple users.

We describe the design and implementation of *Tralfamadore*, an offline dynamic analysis tool that borrows ideas from streaming databases [30] to treat a single execution trace as a shared resource that may be used by a community of developers to understand program behaviour. Tralfamadore explores the opportunities that exist where large reference executions, such as regression test suites or samples of production execution environments, are treated as archival recordings. Our system provides tools to map the low-level details of these recordings up to the source-level semantics used by developers. This approach allows the cost of analysis to be amortized over many users and related queries, making considerably more complex forms of analysis practical.

In this manner, Tralfamadore allows developers to better understand the execution dynamics of the software they are working on: they may explore sweeping queries, like "*show me a histogram of all parameters ever passed to this function,*" "*summarize all execution stacks that have ever invoked this line of code,*" or "*summarize the path through the kernel taken by all received UDP packets,*" in order to quickly get a sense of how complex system code behaves. Furthermore, as the analyses performed by the system are based on the composition of an extensible set of analysis operators, we demonstrate how new analyses can be written, often without requiring a low-level understanding of the nuances of a specific architecture or instruction set.

Our work can be viewed as an extension of recent results that decouple dynamic analysis from live execution [10, 27]. However, rather than simply running traditional analysis against a system recording, we explore the broader set of analyses that are possible across the entirety of an execution trace, and tackle some of the consequent challenges, such as indexing execution for efficient query evaluation. Tralfamadore allows the developer to engage in a dialogue with a system's execution, bridging the gap between the

VEE'12, March 3–4, 2012, London, England, UK.
Copyright © 2012 ACM 978-1-4503-1175-5/12/03. . . $10.00

intention expressed in the source code and the *experience* of actual execution.

2. Goals

Tralfamadore aims to allow developers to understand large, complicated code bases that are difficult to analyze with existing tools. This problem produced four specific design goals for our tool:

2.1 Treat Execution as an Artifact

Traditional dynamic analysis frameworks such as Pin [20] or Valgrind [24] are *online* tools, meaning that they analyze a program as it executes. In Tralfamadore, the execution of the system being analyzed is first recorded and stored persistently, and all analyses are performed afterwards on this static data set. Tralfamadore analyzes executions as opposed to analyzing programs as they execute.

Running dynamic analyses against a static data set has multiple advantages, such as enabling new analyses not foreseen at the time of recording. This approach allows for repeated analysis over the same execution, providing consistency between each run, an important asset for developers using dynamic analysis in the cyclical process of understanding or debugging complex pieces of code such as an operating system kernel. Another advantage is that analyses are no longer "stuck in time" and can freely navigate execution by arbitrarily going back in time or jumping ahead. Analyses themselves are also easier to debug; their deterministic behaviour simplifies the task of validating whether an error in the analysis has been fixed or not. An offline approach also helps alleviate the "observer" effect. Although recording may affect the behaviour of the system, once an interesting behaviour has been captured, this behaviour is guaranteed to manifest for all analyses however intrusive they may be.

2.2 Support Whole-System Analysis

A motivation of our system is to help developers understand behavior that occurs at any layer of the software stack. In this paper we focus on kernel execution, but Tralfamadore is able to record the execution of entire systems. As the OS kernel is typically the most privileged, lowest-level software installed on a system, this goal leads to a number of requirements:

First, the traces should be *comprehensive*, meaning that it should be possible to record the execution of an entire system, including the execution of its operating system kernel and of all its applications. Traces should be captured at the hardware level, underneath the operating system, to ensure a complete visibility of the system's execution. In this paper we focus on kernel execution, but Tralfamadore is able to record the execution of entire systems.

Second, to capture the execution of unmodified systems requires recording execution *transparently* without explicitly changing the target system. Further, being transparent requires Tralfamadore to handle dynamically generated and self-modifying code without the need to be explicitly notified by the system being recorded.

Third, the information recorded should be *complete*, meaning that the recording should contain sufficient information for analyses to inspect the register and memory state of the system at an instruction granularity. Section 4.2 describes how Tralfamadore records execution and provides more details on how it meets these three requirements.

2.3 Bridge the Semantic Gap

Doing dynamic program analysis on an executing binary inherently introduces a *semantic gap*. On one side, the CPU executes a stream of instructions which update the state of the processor and memory, and on the other side, developers expect to be able to reason about a running program at the level of source code. This semantic gap makes developing new analyses challenging. An analysis writer needs to understand many low-level aspects of execution, such as how arguments are passed on function calls, or how memory is allocated from the heap. This effort can be considerable, and requires domain expertise over the instruction set, application binary interface (ABI), etc. This gap widens with operating system code, which does not provide standardized and stable programming interfaces such as POSIX or the C standard library.

```
int deliver_skb(struct sk_buff *skb,
      struct packet_type *pt,
      struct net_device *dev)
{
  atomic_inc(&skb->users);
  return pt->func(skb, skb->dev, pt, dev);
}
```

Figure 1. Packet delivery via type-specific function pointer from the protocol table

Unfortunately, existing dynamic analysis frameworks force analysis writers to constantly reinvent the wheel; low-level aspects of execution must be re-learned for every new analysis. A good example of this problem is Valgrind [24], which includes two different race detectors, and to which engineers at Google recently added a third one.[1] Although all three detectors differ in the algorithm used to identify races, they each reimplement instrumentation to extract the same information (lock acquisition/release, accesses to heap memory). The required instrumentation is orthogonal to the analysis used and could be shared across implementations if Valgrind made it easy to do so.

2.4 Provide Interactive Results

Finally, as the previous goals lead to a system that performs deep and complex analysis over large volumes of execution data there is a risk that analysis may take a long time to complete. Tralfamadore aims to enable highly interactive exploration of trace data, and so we desire that the system provide interactive results so that developers can get answers as quickly as possible. To achieve this, Tralfamadore provides mechanisms to generate trace indices that subsequent analyses can use to only read the relevant portions of a trace. Many analyses end up reading a small fraction ($< 1\%$) of the trace. The Tralfamadore framework is based on a dataflow model that is well suited to produce results in an online manner. Analyses can return partial results as soon as they are available. This is advantageous for analyses that take a long time to complete because they need to read a substantial fraction of the trace. All the analyses described in this paper can return partial results within seconds, although some of them take much longer to complete.

3. An Example: Linux Networking

The Linux network stack is an illustrative example of the challenges faced by developers in understanding unfamiliar code. This code base has a number of features that contribute to the difficulty of understanding it.

First, it is very large. According to SLOCCount[2], there are 342,536 lines of code in the net subdirectory of the Linux kernel (version 2.6.24). Restricting our attention to the most relevant subcomponents (ipv4, netfilter, core, sched, and ethernet) reduces the count to 113,110 lines. Second, it spans all the layers of kernel abstractions from system calls through device drivers. Third, it uses dynamic control flow techniques that are challenging for static tools to understand, such as function pointers (Figure 1 shows one ex-

[1] http://code.google.com/p/data-race-test/wiki/ThreadSanitizer

[2] http://www.dwheeler.com/sloccount/

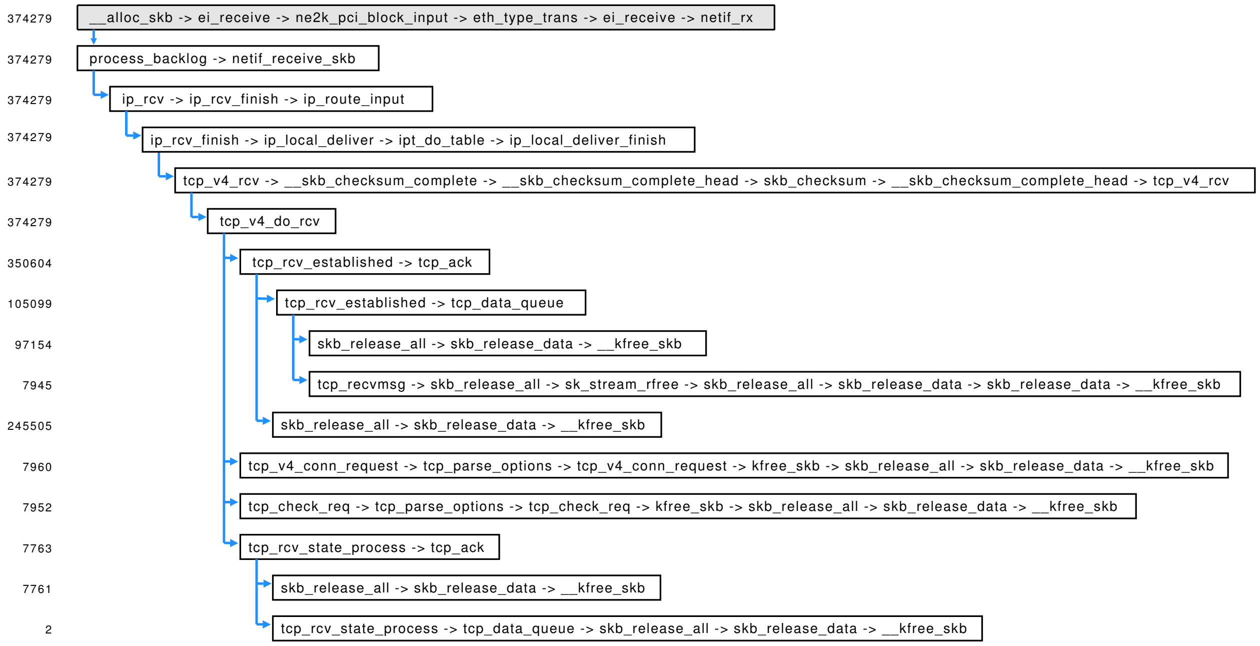

374279
__alloc_skb -> ei_receive -> ne2k_pci_block_input -> eth_type_trans -> ei_receive -> netif_rx
374279
process_backlog -> netif_receive_skb
374279
ip_rcv -> ip_rcv_finish -> ip_route_input
374279
ip_rcv_finish -> ip_local_deliver -> ipt_do_table -> ip_local_deliver_finish
374279
tcp_v4_rcv -> __skb_checksum_complete -> __skb_checksum_complete_head -> skb_checksum -> __skb_checksum_complete_head -> tcp_v4_rcv
374279
tcp_v4_do_rcv
350604
tcp_rcv_established -> tcp_ack
105099
tcp_rcv_established -> tcp_data_queue
97154
skb_release_all -> skb_release_data -> __kfree_skb
7945
tcp_recvmsg -> skb_release_all -> sk_stream_rfree -> skb_release_all -> skb_release_data -> skb_release_data -> __kfree_skb
245505
skb_release_all -> skb_release_data -> __kfree_skb
7960
tcp_v4_conn_request -> tcp_parse_options -> tcp_v4_conn_request -> kfree_skb -> skb_release_all -> skb_release_data -> __kfree_skb
7952
tcp_check_req -> tcp_parse_options -> tcp_check_req -> kfree_skb -> skb_release_all -> skb_release_data -> __kfree_skb
7763
tcp_rcv_state_process -> tcp_ack
7761
skb_release_all -> skb_release_data -> __kfree_skb
2
tcp_rcv_state_process -> tcp_data_queue -> skb_release_all -> skb_release_data -> __kfree_skb

Figure 4. TCP packet receive

```
int netif_rx(struct sk_buff *skb)
{
  struct softnet_data *q;
  q = &__get_cpu_var(softnet_data);
  __get_cpu_var(netdev_rx_stat).total++;
  __skb_queue_tail(&q->input_pkt_queue, skb);
  return NET_RX_SUCCESS;
}

int process_backlog(..., int quota)
{
  struct softnet_data *q;
  q = &__get_cpu_var(softnet_data);

  do {
    struct sk_buff *skb;
    skb = __skb_dequeue(&q->input_pkt_queue);
    netif_receive_skb(skb);
  } while (++work < quota);

  return work;
}
```

Figure 2. Top/bottom half network processing

ample of this, the packet delivery function, which dispatches the packet through a type-specific function pointer). Fourth, the functionality of the network stack is split in accordance with the standard Linux top/bottom driver architecture: the "top half" executes in interrupt context, which must complete as quickly as possible. It simply receives a packet from a device driver and queues it. Actual packet processing (the "bottom half") is deferred until interrupts have been serviced. Figure 2 shows the code used by network card interrupt handlers to queue received packets and the processing function that runs asynchronously to dequeue packets to push them up the protocol stack. This causes standard call-flow analysis to be unable to capture the end-to-end processing of packets from arrival to final delivery.

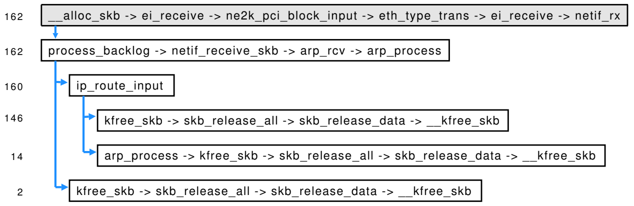

162
__alloc_skb -> ei_receive -> ne2k_pci_block_input -> eth_type_trans -> ei_receive -> netif_rx
162
process_backlog -> netif_receive_skb -> arp_rcv -> arp_process
160
ip_route_input
146
kfree_skb -> skb_release_all -> skb_release_data -> __kfree_skb
14
arp_process -> kfree_skb -> skb_release_all -> skb_release_data -> __kfree_skb
2
kfree_skb -> skb_release_all -> skb_release_data -> __kfree_skb

Figure 3. ARP packet lifetime through the network stack

Static analysis tools that perform call flow analysis cannot deal with this level of indirection, but even if they could, constructing a static call graph only tells part of the story. A large portion of system code is there to handle errors and rare corner cases. Static analysis cannot guide a developer to concentrate on the important or most commonly used functionality of the network stack, nor can it identify unused or outlying code paths.

The network stack is inherently data-driven and has a single data structure (the packet) at its core. Developers who want to understand the stack typically want to begin with understanding how a specific subset of packets flow through it. What if a developer could point to a function in the source code and ask, "Where are packets processed by this function coming from and where are they going to?" The answer to this question for the function `arp_rcv` is illustrated in Figure 3 which shows a tree produced by our tool of the functions that access every packet which passes through `arp_rcv`, from its allocation to is eventual release. Function names in the first, shaded text box are executed in device interrupt context, while subsequent processing occurs in a `soft_irq`. Numbers to the left of the figure indicate the number of packets that take the indicated path through the code. Looking at the figure, the developer can quickly learn that packets are allocated by the function `__alloc_skb`, are operated on at the ARP level primarily by the function `arp_process`, and are eventually freed by `__kfree_skb`.

Having understood this simple flow for ARP packets, the developer might then ask about packets processed by another function, perhaps for a more complex protocol such as TCP. The flow of packets going through the function `tcp_v4_do_rcv` is shown in Figure 4. The graph is much larger, representing the increased complexity of TCP processing relative to ARP.

The figures in this section were generated from a trace of a virtual machine running the Apache web server on Linux. The data used to generate these figures is obtained with a new analysis we have dubbed *Heap Slicing* which we describe in Section 5.4, after we have explained the overall architecture of our platform.

4. The Trace Analysis Engine

Figure 5 illustrates the Tralfamadore architecture. Analysis starts with an execution trace, which is captured at the virtual machine level and stored as a database which the analysis engine uses to answer queries. User queries are constructed out of simple modules under a dataflow model similar to network packet processing engines such as Click [16] and Bro [28]. In this paradigm, data is processed by single-function components, which are called *operators* in Tralfamadore. These operators are connected as a DAG, where each operator consumes a stream of records from one or more upstream operators and produces a stream of records for downstream operators. Data traverses the graph of operators, *flowing* from the leaves to the root of the tree.

Tralfamadore analyses execute by parsing the trace into a *stream* of records called *annotations* corresponding to the execution of machine-level instructions and the occurrence of events such as interrupts or page faults. The stream is passed through an analysis-specific configuration of operators that progressively augment the stream with semantically richer annotations corresponding to higher-level events such as function calls, lock uses, or heap allocations. Analyses can be expressed succinctly in terms of higher level annotations instead of dealing directly with a CPU-level trace.

The current implementation of the Tralfamadore analysis engine consists of approximately 5000 lines of C code and 12000 lines of OCaml code. The C language is used to implement performance critical code such as the trace parser and the instruction disassembler. All of the operators and analyses are written in OCaml. OCaml is a garbage-collected, statically typed functional language with imperative and object-oriented features. We choose OCaml because many of its features such as variant types and pattern matching map naturally to the tasks accomplished by Tralfamadore operators.

4.1 Primitives

Operators are the main functional abstraction in Tralfamadore. Each operator is designed to recognize specific patterns in the stream corresponding to the occurrence of events and produce annotations that describe these events. The type of an operator is solely defined by the annotations it produces. This encapsulation makes it easy to replace the implementation of an operator with a different one.

Annotations are the basic unit of information in the system. They are typed and parameterized, and operators are able to define new annotations that they will produce for downstream operators. Annotations are logically timestamped to provide an ordering between annotations which is needed when merging multiple annotation streams. The trace parser operator reads the raw trace on disk and produces *basic block* and *event* annotations, which are the most basic form of annotations used in the system. Section 5 describes some of the current stream operators and the annotations that they use. We are actively developing new operators in order to provide support for other operating systems and the analysis of user-level applications.

A **stream**, then, is an interface across which operators may send and receive sequences of annotations. As mentioned above, the flow of annotations in the system is driven by a "pull" from the sink operator in the DAG rather than a "push" from the trace itself. This allows operators to elect to work with specific, small regions of the trace. The programmatic idiom for streams in our system is that of an infinite lazy list, whose contents are materialized on demand.

4.2 Tracing

Almost all operators consume annotations from other operators in order to produce semantically richer annotations. The most important exception is the *trace operator*, which consumes the trace itself to produce annotations representing changes to the state of the system at the (virtual) hardware level. The trace is captured by running the target system on a modified version of QEMU [5]. QEMU is a fast whole-machine emulator that uses dynamic binary translation to emulate a guest instruction set architecture on a (possibly different) host architecture. QEMU breaks guest instructions into a series of RISC-like micro-operations which are fed into a code generator to produce native code that executes directly on the host. We made straightforward additions to these micro-operations to log their changes to emulated guest state as they execute. The resulting log is a faithful recording of every change to guest state, detailed enough that the exact state of the machine at any point in its execution can be recreated later. The modifications we have made to support trace generation are based on QEMU version 0.9.1 and consist of approximately 2200 lines of C code.

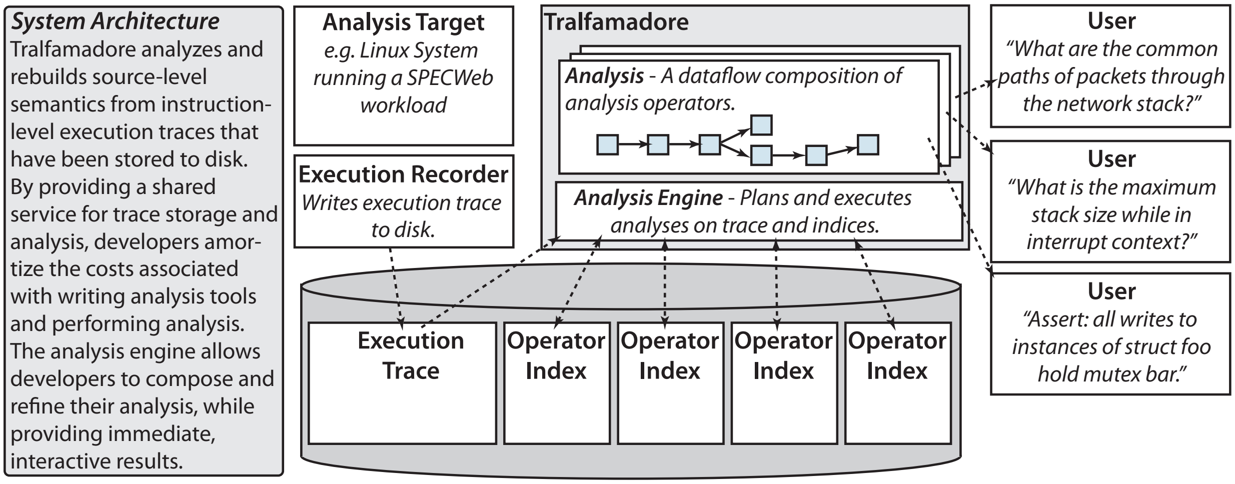
In order to support arbitrary analyses, the execution trace must record every change to the state of the machine as it runs. This includes every instruction executed, every exception, every interrupt, and all changes to register and memory, whether due to instructions or I/O.

```
c01373e8 sti            EFL=00000206
c01373e9 nop 0x0(%eax,%eax,1)
E[20,0000,c01373f1]     STL[cd903f74]=206 STL[cd903f70]=60
                        STL[cd903f6c]=c01373f1 R[ESP]=cd903f6c
                        S[CS]=0060,00000000,ffffffff,00cf9a00
                        BR=c010859c EFL=00000006
c010859c push $0xffffffff STL[cd903f68]=ffffffff R[ESP]=cd903f68
c010859e jmp 0xc0108ddc   BR=c0108ddc
c0108ddc cld            EFL=00000006
c0108ddd push %fs       STL[cd903f64]=000000d8 R[ESP]=cd903f64
c0108ddf push %es       STL[cd903f60]=0000007b R[ESP]=cd903f60
c0108de0 push %ds       STL[cd903f5c]=0000007b R[ESP]=cd903f5c
c0108de1 push %eax      STL[cd903f58]=00b77000 R[ESP]=cd903f58
...
c0108de7 push %ebx      STL[cd903f40]=00000002 R[ESP]=cd903f40
c0108de8 mov $0x7b,%edx   R[EDX]=0000007b
c0108ded mov %edx,%ds     S[DS]=007b,00000000,ffffffff,00cff300
```

Figure 6. Trace sample.

Figure 6 is a concrete example of the level of detail captured by our trace collector. At the start of the section, the CPU program counter is at `0xc01373e8`, which contains the `sti` instruction that causes interrupt delivery to be enabled (setting bit 9 of the `EFLAGS` register). Upon execution of this instruction, an interrupt (`0x20`, the timer interrupt) is immediately delivered. As part of interrupt delivery, the CPU saves essential state on the stack, disables interrupt delivery, and branches to an interrupt dispatch function. The dispatcher saves register values on the stack to free the registers for use by the interrupt handler it is about to call.

As is probably apparent from Figure 6, this approach produces an enormous amount of trace data, and slows execution significantly (see Section 7 for details). For a system like ours, which is designed as a developer tool and in which a single trace can be reused for large numbers of analyses, the benefits can be worth the costs. For applications that demand less overhead during execution (e.g., debugging, security, or forensics), it is possible to re-



System Architecture

Tralfamadore analyzes and rebuilds source-level semantics from instruction-level execution traces that have been stored to disk. By providing a shared service for trace storage and analysis, developers amortize the costs associated with writing analysis tools and performing analysis. The analysis engine allows developers to compose and refine their analysis, while providing immediate, interactive results.

Analysis Target
e.g. Linux System running a SPECWeb workload
Execution Recorder
Writes execution trace to disk.
Tralfamadore
Analysis - A dataflow composition of analysis operators.
Analysis Engine - Plans and executes analyses on trace and indices.
Execution Trace
Operator Index
Operator Index
Operator Index
Operator Index
User
“What are the common paths of packets through the network stack?”
User
“What is the maximum stack size while in interrupt context?”
User
“Assert: all writes to instances of struct foo hold mutex bar.”

Figure 5. Tralfamadore Architecture

duce recording cost by separating it from trace generation, using deterministic virtual machine record and replay [11]. These logs are small, and recording them imposes only modest performance costs. As an event log, the recording contains very little information about execution, making it unsuitable for direct analysis. However, it provides enough information to regenerate the entire execution history in a subsequent, offline replay phase during which the full execution trace can be produced [38]. We are currently working on adding support for deterministic record/replay in QEMU and in Xen. Our goal is to be able to record execution in Xen and replay it in QEMU, similarily to Aftersight [10], to generate traces.

4.3 Trace Analysis

Figure 7 illustrates how operators gradually reconstruct execution semantics when streaming a CPU-level trace through the configuration of operators shown in Figure 8. The left hand side of Figure 7 shows an excerpt of raw trace containing basic block (BBlk) annotations. These annotations are generated by the *Trace Parser* operator. The stream passes through the *Invoke* operator which augments the stream with *Invoke* and *Return* annotations corresponding to function calls and return statements. A third operator (*Alloc*), uses the *Invoke* annotations to identify calls to `__kmalloc()`, and further augments the stream with heap allocation annotations. The *Alloc* annotation shown on the right hand side indicates the allocation of a block of 1024 bytes at address `0xdfcc5800`. Subsequent operators interested in heap allocations do not need to be concerned with identifying calls to `__kmalloc()`, but can simply use the *Alloc* annotation directly without knowing how it was generated.

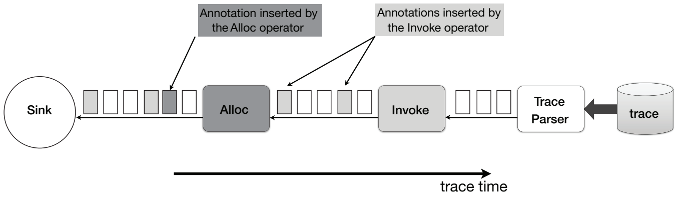
In addition to gradually bridging the semantic gap, the dataflow architecture encourages the design of analysis that are *composable* and *reusable*. Composable means that analyses can be easily structured as a combination of simple single-function operators. This design also makes it easy to reuse these operators across multiple analyses, saving developers from the burden of constantly having to re-understand low-level aspects of execution.

4.4 Caching and Indexing

Many analyses only need to look at small portions of the trace to extract the information they need. A good example is extracting the value of an argument for all calls to a specific function. It is sufficient to revisit the trace at all call sites and inspect the register or memory state to extract the argument values. Analyses could execute much faster if they had the ability to only read the relevant portions of a trace.

```
BBlk(2):
c01b0623: mov  $0x000000d0  -> %edx : R[edx]=000000d0
c01b0628: call 0xfffe7228 %esp -> %esp (%esp) : ST32[f755bea4]=c01b062d
          R[esp]=f755bea4 BR=c0197850
                              Invoke (__kmalloc, 0xf755bea4)
BBlk(8):
c0197850: sub  $0x1c %esp -> %esp : R[esp]=f755be88
c0197853: cmp  %eax $0x00000800:
c0197858: mov  %ebx -> 0xc(%esp): ST32[f755be94]=ff
c019785c: mov  %esi -> 0x10(%esp): ST32[f755be98]=100
c0197860: mov  %edi -> 0x14(%esp): ST32[f755be9c]=f755bf0cc0197864:
          mov %ebp->0x18(%esp): ST32[f755bea0]=dfda0600
c0197868: mov  %edx -> 0x8(%esp) : ST32[f755be90]=d0
c019786c: jcc  0x00000099 : FL=00000287
...
BBlk(7):
c01978ce: mov  %ebx  -> %eax : R[eax]=dfcc5800
c01978d0: mov  0xc(%esp) -> %ebx : LD32[f755be94]=ff R[ebx]=000000ff
c01978d4: mov  0x10(%esp) -> %esi : LD32[f755be98]=100 R[esi]=00000100
c01978d8: mov  0x14(%esp) -> %edi : LD32[f755be9c]=f755bf0c R[edi]=f755bf0c
c01978dc: mov  0x18(%esp) -> %ebp : LD32[f755bea0]=dfda0600 R[ebp]=dfda0600
c01978e0: add  $0x1c %esp -> %esp : R[esp]=f755bea4
c01978e3: ret  %esp (%esp) -> %esp : LD32[f755bea4]=c01b062d R[esp]=f755bea8
          BR=c01b062d FL=00000292
                              Return (0xf755bea4)
                                                Alloc (0xdfcc5800, 1024)
```

Figure 7. Reconstruction of execution semantics using progressively higher-level annotations.



Annotation inserted by the Alloc operator
Annotations inserted by the Invoke operator
Sink
Alloc
Invoke
Trace Parser
trace
trace time

Figure 8. Operator configuration.

To efficiently support this sort of analysis, Tralfamadore operators can be configured to create persistent caches of the annotations they generate. Subsequent analyses can use cached annotations as indices to map annotations or annotation attributes to interesting positions in the trace. These positions can be used as starting points to examine a trace, or used to identify interesting slices of a trace, such as the range of a trace where a specific thread was running. Operators that support this mechanism export a query interface that lets analyses instantiate an operator that will produce the subset of annotations matching the query. This is conceptually similar to

Operator	Description
Trace Parser	Parses the raw trace file. Produces: **_BBlk_**_(pc, instructions, side effects)_, **_Event_**_(vector, error code, pc, side effects)_
Context	Identifies independent contexts of execution (threads, interrupts). Depends: **_BBlk_**_(side effect_ = `store[TSS.esp0]|`_mnemonic_ = `sysenter|iret|...`_)_, **_Event*_** Produces: **_Context_**_(act id, stack addr,_ `Entry|Exit|Switch`_)_
Invoke	Identifies function call and return points. Depends: **_BBlk_**_(pc = function addr | instr.mnemonic =_ `ret`_)_ Produces: **_Invoke_**_(pc, stack pointer)_, **_Return_**_(stack pointer)_
Alloc	Tracks the allocation and release of objects on the heap. Depends: **_Context*_**, **_Invoke*_**, Produces: **_Alloc_**_(ptr, size, pc)_, **_Release_**_(ptr)_
MemTrace	Tracks memory access. Depends: **_BBlk_**_(side effect_ = `load[*]|store[*]`_)_ Produces: **_MemTrace_**_(pc, address, access size,_ `R|W`_)_
HeapTrace	Tracks accesses to objects allocated on the heap. Depends: **_MemTrace*_**, **_Alloc*_** Produces: **_HeapTrace_**_(pc, ptr, offsetm access size,_ `R|W`_)_

Table 1. Some of the current library of streaming operators available in Tralfamadore.

a *select* operation in a relational database where a subset of the rows—those that match the selection criteria—are returned.

5. Operator Library

Whole-system analysis faces unique challenges in mapping CPU-level execution to source-level semantics. Our approach provides a framework in which new analyses can be composed from existing operators that refine low level trace annotations into progressively higher-level semantics. While our focus with the system to date has been in studying the Linux operating system on x86 hardware, we have also begun work on Windows support, as well as user-level analysis. Extending the system in these directions is largely a matter of developing new operators. Tralfamadore currently hosts a library of about twenty operators, a subset of which are described in Table 1.

Operators are intended to be standalone sections of code that embed a specific aspect of analysis. Lower-level operators include detailed knowledge about the hardware and how it is used by the OS, while higher-level operators encode details about how the OS behaves, for instance with regard to its heap management. The remainder of this section describes three of the more complex of the fundamental operators in additional detail.

5.1 Execution Contexts

One of the primary responsibilities of an operating system is to schedule independent threads of execution onto shared physical CPUs. Since the trace is a record of instructions at the CPU level, these threads of control will appear to be arbitrarily interleaved. Yet many modes of analysis (the simplest is call flow extraction) operate on single execution flows. Therefore, we provide a *Context* operator for demultiplexing the trace. Tracking contexts in operating systems kernels is challenging, due to the asynchronous, interrupt-driven environment in which they operate: a flow of execution may be suspended and resumed for a number of reasons, including explicit context switches, interrupts (which may be nested), and exceptions. Identifying these transitions requires an understanding of both the physical architecture and how a given OS uses that architecture in its implementation. Specifically, the *Context* operator needs to track three things: the start and end of interrupt and fault handlers, system calls, and the occurrence of context switches from one thread to another.

The execution of interrupt and exception handlers (including system calls that use software interrupts to enter the kernel) can be identified by Event annotations in the trace. At these points, the current flow of execution is suspended and control jumps to an event handler. An *iret* instruction marks the end of handler execution and the resumption of the original execution context. System calls may also be invoked through fast entry instructions that do not generate events; their boundaries are identified as pairs of *sysenter/sysexit* Instruction annotations in the stream.

There are multiple ways to identify software execution contexts, with different trade-offs. In order to pair a thread resumed by the operating system scheduler with its previously suspended execution the *Context* operator needs a mechanism to identify each thread in the system. One simple way to do this would be to track the invocation of the operating system stack switching routine such as `__switch_to` on Linux, using its argument to identify threads. We currently use a more generic approach (that works for both Linux and Windows) which is to track updates to the esp0 field of the Task State Segment (TSS). This field indicates the base address of the kernel stack for the task about to run. This address uniquely identifies a thread over its lifetime and thus serves as a suitable context identifier.

The *Context* operator tags each flow by emitting an annotation whenever a flow of execution is suspended or resumed (whether due to a software context switch or a hardware interrupt). The first attribute of the annotation is an activation ID that uniquely identifies a flow (such as the invocation of a system call or interrupt handler) over its lifetime. The second attribute is the base address of the kernel stack, which can be used to group all system calls executed by the same thread.

To deal with nested interrupts, the operator maintains a stack of live activations for each thread it has seen. When an interrupt occurs, the operator emits an annotation with a new activation ID and pushes that ID on the stack. When the handler terminates, the operator pops its stack and emits an annotation with the ID of the resumed activation. Similarly, on a software context switch, the operator emits an annotation indicating the activation and stack address of the resumed thread.

Analyses using annotations from the *Context* operator require no specific knowledge of Linux or the x86 architecture. They may simply use the annotations to identify the regions of trace that comprise a single activation or thread of execution.

5.2 Invocations

The Invoke operator produces annotations that simplify the tracking of call flow for downstream operators; compiler optimizations such as tail call elimination can make this challenging. The operator produces two annotations: *Invoke* and *Return*. The *Invoke* annotation includes the function being called and the value of the stack pointer at the time of the call. The *Return* annotation includes the value of the stack pointer just before the return address is consumed by a ret instruction. The stack pointer can be used as a key to pair matching function calls and returns. In the case of tail call optimization where a single ret instruction will unwind multiple calls, a *Return* annotation will match more than one *Invoke* annotation.

Inline functions cannot be tracked in this way, since they are not explicitly called. We do, however, provide limited support (restricted to control flow) for inline functions based on debugging information.

5.3 Allocations

The *Alloc* operator tracks the allocation of objects on the heap and issues annotations whenever an object is allocated or freed. It contains domain-specific knowledge of the allocation functions being used. The allocation annotation contains the address of the object, its size and the address of the caller to the allocation function.

The biggest difficulty with tracking allocations is that with most allocators the parameters used to determine the size — either the size itself or a pointer to a memory pool — are passed on invocation, but the pointer to the object is only available on return. The *Alloc* operator takes advantage of the stack pointer matching provided by the *Invoke* operator's annotations to match the object with its size.

5.4 Putting It Together: Heap Slicing

The figures in Section 3 were generated by extracting *heap slices* from a trace of a virtual machine running the Apache web server on Linux. Heap slicing is a new analysis that extracts the set of statements that have touched a set of heap objects between their allocation and release. It produces a representation of where and how these objects are used throughout their lifetime.

The goals of heap slicing are different than with program slicing [4, 36], a program analysis technique which extracts program statements that have affected the value of a variable at a specific point in a program. This set of statements, called a program slice, helps developers focus on the relevant portion of a program when trying to understand how and why a variable holds a certain value. The goal of heap slicing is more holistic as it is to provide a view of how an entire data-driven subsystem operates at runtime.

The inputs to heap slicing are a function and one of its arguments which should be a pointer to a heap object. The analysis executes in two passes over a trace. In the first pass, the value of the pointer is extracted at all points in time where the function is called. This pass uses the function index to only visit relevant call sites, skipping most of the trace. The (timestamp, pointer) tuples extracted in this first pass are cross-referenced with a heap allocation index to determine the allocation and release time for each pointer/heap object.

In the second pass, for every object, the analysis scans the trace from allocation to release to extract all accesses to the object throughout its lifetime. The result is an access trace for each object. The results are forwarded to the frontend where they are summarized in a tree, similar to the ones shown in Figure 3 and 4, where each access trace is inserted from the root down using longest prefix matching. The root of the tree corresponds to the allocation function and the leaf nodes are deallocation functions. A given analysis can produce more than one graph if the objects were allocated by more than one function.

This two pass approach is only possible with an offline framework such as Tralfamadore where multiple analyses or analysis passes can execute over the same persisted execution. From an execution point of view, the analysis effectively goes back in time from the point in time where the selected function is called to the point where the object is allocated. Heap slicing could be implemented in an online manner but such an implementation would be highly inefficient for two reasons. First, the analysis, unaware of where objects are allocated and released, would need to examine the entire execution. Analysis built with Tralfamadore can leverage indices to only look at sections of trace containing live objects. Second, it would have to track memory accesses to all heap objects, and then discard objects that were never manipulated by the function of interest. Tralfamadore only needs to track accesses to the set of objects that will actually be part of the slice, reducing the overhead of the analysis.

```
1  module HeapTraceOp  = struct
2
3   (* This operator acts as a filter,
4      removing all annotations not part of an object heap trace. *)
5
6   let rec next st () =
7     (* Pull the next annotation from the upstream operator *)
8     let a = st.m.M2Op.next () in
9     match a.Annot.attr with
10    | ‘Alloc alloc -> (
11      st.live_obj <- add_to_live_obj st.live_obj alloc;
12      a
13    )
14    | ‘Release rel -> (
15      st.live_obj <- del_from_live_obj st.live_obj rel;
16      if live_obj_is_empty st.live_obj then (
17        (* If there is no live obj, seek to the next allocation. *)
18        try st.m.M2Op.seek_1 () with EOF -> ()
19      );
20      a
21    )
22    | ‘MemTrace mt -> (
23      (* If this memtrace annotation corresponds to an access to a live
24         heap object, produce a heap trace annotation. *)
25      try
26        (* Find the allocation matching the access *)
27        let alloc = find_in_live_obj st.live_obj mt.MemTrace.addr in
28        let off = Guest32.sub addr alloc.Alloc.base in
29        ‘HeapTrace { base = alloc.Alloc.base; off = off;
30                     sz = mt.MemTrace.sz; access = mt.MemTrace.access }
31      with Not_found -> next st ()
32    | _ -> next st ()
33
34  ...
35
36  end
```

Figure 9. HeapTrace operator.

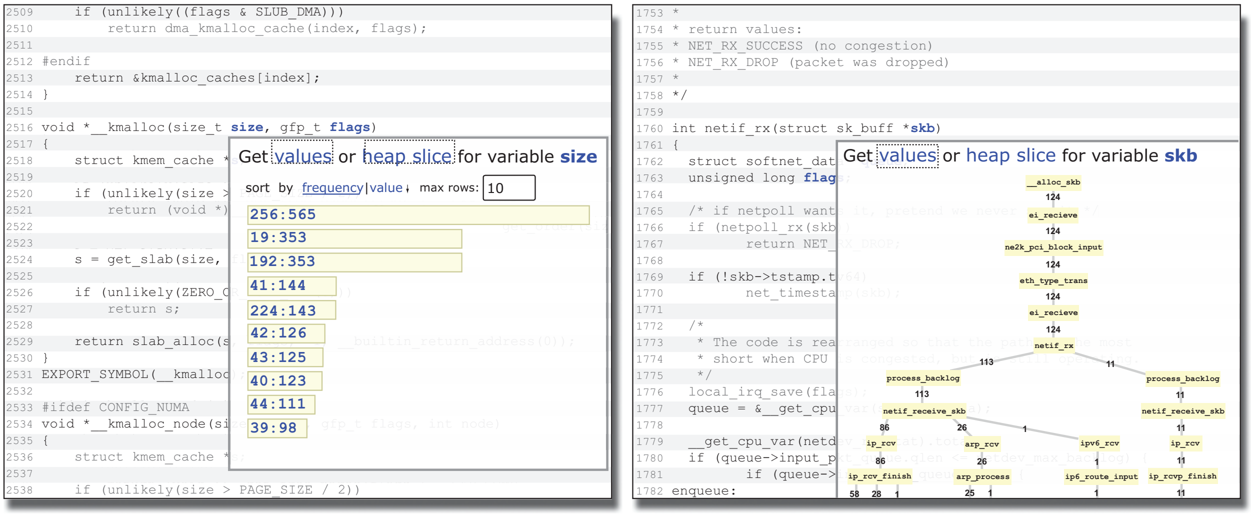
Figure 9 shows the core of the *HeapTrace* operator which extracts the object access traces, skipping sections of trace without live objects. This example shows how an operator reuses existing operators to compose a new analysis. The operator writer needs not to be concerned with specific knowledge of the Linux kernel heap allocation functions or which x86 instruction touches memory. This knowledge is abstracted away by using the *Alloc* and *Release* annotations (Line 10 and 14 respectively) produced by the *Alloc* operator, and the *Memtrace* annotations (Line 22) produced by the *MemTrace* operator.

6. Example Applications

Tralfamadore preserves a complete execution as a data set in order to make it possible to explore it interactively in the same way that developers interrogate source repositories. To support this usage, and to demonstrate the types of analyses that Tralfamadore facilitates, we have built several applications on the platform. In this section, we present our prototype execution browser, which integrates interactive execution queries directly into a conventional source code browser using a simple and intuitive interface. We follow this with a series of more complex applications that highlight the development advantages of our data-driven processing architecture, in which simple pipelines of reusable operators combine to produce powerful analyses.

6.1 Understanding Execution

Reasonably simple applications of static analysis are often used to assist developers in navigating large bases of source code. Editors have "tag" facilities, and often also allow developers to use a search facility that is tied to a language-specific parser in order to find things like the declaration of a specific variable or the definition



```
    if (unlikely((flags & SLUB_DMA)))
        return dma_kmalloc_cache(index, flags);

#endif
    return &kmalloc_caches[index];
}

void *__kmalloc(size_t size, gfp_t flags)
{
    struct kmem_cache *

    if (unlikely(size >
        return (void *)


    s = get_slab(size,

    if (unlikely(ZERO_O
        return s;

    return slab_alloc(s
}
EXPORT_SYMBOL(__kmalloc

#ifdef CONFIG_NUMA
void *__kmalloc_node(si
{
    struct kmem_cache *

    if (unlikely(size > PAGE_SIZE / 2))
```

Get values or heap slice for variable **size**

sort by frequency|value max rows: 10

```
256:565
19:353
192:353
41:144
224:143
42:126
43:125
40:123
44:111
39:98
```

```
*
* return values:
* NET_RX_SUCCESS (no congestion)
* NET_RX_DROP (packet was dropped)
*
*/

int netif_rx(struct sk_buff *skb)
{
  struct softnet_dat
  unsigned long flag

  /* if netpoll want
  if (netpoll_rx(skb
        return NET_

  if (!skb->tstamp.t
        net_timesta

  /*
   * The code is rea
   * short when CPU
   */
  local_irq_save(fla
  queue = &__get_cpu

  __get_cpu_var(netd
  if (queue->input_p
        if (queue->
enqueue:
```

Get values or heap slice for variable **skb**

Figure 10. Live data value and heap slice analyses from the source understanding tool. (Section 6.1)

of some function. Dynamic techniques have also been applied to help developers understand how source behaves under execution by exposing details such as profiling and coverage information. With Tralfamadore, we felt that a significantly more powerful tool could be provided for understanding program execution. We have developed a web-based source navigator that maps interactive analysis tools directly onto the source code.

In a previous paper [18], we described an early prototype of this source navigator that maps binary trace information into source-level call and control flow graphs. With this initial support, a function in source can be annotated with a graph that shows the specific unique control flow paths that travel through it, and allows developers to choose between focusing on the common case, outliers, or unexecuted code, depending on their interest.

Figure 10 shows the interface provided by our source understanding tool for two new data-oriented analyses. From the source browser, a developer may select a function parameter and request a histogram of the values of that parameter across all calls to the function, or, in the case of pointers to heap objects, they may generate a data flow graph that summarizes all accesses to the set of objects passed to that function. Partial results are streamed into the browser over AJAX as soon as they become available, and the result set is continuously updated as the query is processed.

6.1.1 Argument value distribution

The left-hand side of Figure 10 shows an online query in which a user browsing the source to the Linux `__kmalloc` memory allocator has become curious about the size of allocation requests observed during a trace. She clicks on the `size` argument to the function and is presented with a pop-up window in which she can select from one of two variable queries. In this case, she chooses the *values* option. Immediately, a histogram appears showing the sizes observed in the trace. At the point seen in the figure, the query has been running for 1 second and shows that the most commonly requested object size so far is 256 bytes.

6.1.2 Heap Slicing

On the right of Figure 10 is a screen capture of a heap slicing query in progress (figures in Section 3 were generated by this tool and adjusted for presentation). In this case, the user has clicked on the `skb` parameter of the `netif_rx` function and chosen the *heap slicing* option. The heap slice of all packets passing through `netif_rx` begins to form, stabilizing within a few seconds.

This example extends recent work in visualizing data flow in systems code [23], by allowing the dynamic and *interactive* mapping of arbitrary objects in source to a graph illustrating the use of all instances of that object in actual execution. In the future, we hope to extend this tool to allow users to navigate execution by iteratively following this sort of graph across both control flow and data flow representations of execution.

6.2 Retroactive Assertions

The use of assertions is common practice when attempting to debug, understand, or otherwise validate assumptions about the way that a system is behaving. Unfortunately, they typically must be made *a priori*, either by compiling assert statements into source or inserting dynamic probes prior to running a system. VAssert [2] partly solves this problem by allowing assertions on applications running in a virtual machine to be validated during replay, but while they allow the evaluations of assertions to be deferred, the assertions themselves are no more powerful. We now consider the use of Tralfamadore to validate *retroactive assertions* that would be very difficult to write in the traditional style.

6.2.1 Ownership Violation

For simplicity and scalability, applications desire to use as little locking as possible. In Linux, for example, many functions assume that their objects are never accessed in interrupt context. They may use runtime assertions[3] to catch some violations, but it is difficult to be assured that all possible accesses have been guarded.

Tralfamadore makes it straightforward to validate whether such an assumption holds throughout the duration of a trace, and can additionally produce a detailed report about the violation. Here we present a simple *ownership* assertion that detects whenever an object is accessed by contexts other than the one in which it was allocated, which may be checked against all instances of a given object type within the trace. The algorithm is a refinement of the heap slicing algorithm presented in Section 6.1.2: Each object's lifetime is looked up from the allocation index, and the moment of allocation is then cross-referenced with the context index to identify the *allocation context*. The trace can then be scanned linearly from the time of allocation to the point where the object is freed. Whenever the object is touched, the *accessor context* is recorded. If the set of accessor contexts contains multiple

[3] BUG_ON(in_interrupt());

elements, then an ownership violation has occurred. If a violation is detected, a subsequent query can retrieve the necessary information to produce a graph showing the exact sequence of accesses that produced the violation overlayed with the control flow of both the *owner* and *violator*. Figure 11 shows a portion of the report produced by a test case we created, in which an *owner* thread creates an object, and a *violator* running as a timer interrupt handler accesses it. The sequence on the left is an abridged call trace of the owner context, and the one on the right is the call trace of the violator context. The chain of arrows between the two represents the sequence of accesses, including the violating access.

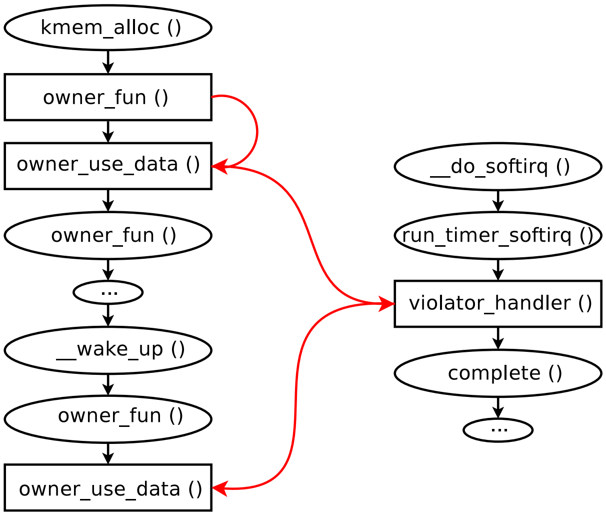


Figure 11. Ownership violation example

6.2.2 Stack Usage

Stack space in the kernel is at a premium: each thread of each process needs its own stack, and the address space is small. For this reason, Linux includes a compilation option to use 4K stacks instead of the default 8K. Unfortunately, stack usage is a highly dynamic property, depending on a combination of control and data (for example, a recursive function's stack usage can vary wildly depending on input). And so while the developers believe that 4K should suffice, and that it would allow many more threads, the default remains at 8K. The kernel developers are washing their hands of the problem, leaving it to the end user to decide whether to take the chance of stack corruption in exchange for better scalability, with no good idea of even how likely that corruption may be.

If the target workload is run under Tralfamadore, the actual amount of stack used can be easily measured. We constructed a simple query that measures the amount of stack in use whenever it changes. Running it on the `kernel` workload described in Section 7 produces the usage distribution shown in Figure 12. This reveals the exact moment at which the stack reaches its largest point (in this case, the maximum size of 2960 bytes was reached 14 times). A query for the call stack at that point returns the stack shown in Figure 13, showing that the cause of the high stack usage was a network interrupt being handled while the kernel was deep in the processing of a *sys_write* system call. Further inspection revealed that all 14 occurrences were at this location, due to the interrupt handler repeatedly processing a backlog of packets. This easily reveals the motivation for the decision of the Linux developers to enable a separate interrupt stack frame when the 4K stack option is enabled.

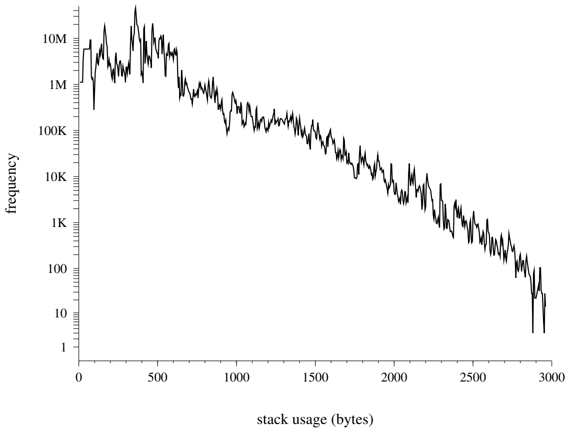


Figure 12. Distribution of stack sizes

```
sysenter_past_esp/sys_write/vfs_write/do_sync_write/
ext3_file_write/generic_file_aio_write/
__generic_file_aio_write_nolock/generic_file_buffered_write/
ext3_write_begin/block_write_begin/__block_prepare_write/
ext3_get_block/ext3_get_blocks_handle/ext3_new_blocks/
__ext3_journal_get_write_access/journal_get_write_access/
do_get_write_access/__get_free_pages/__alloc_pages/
get_page_from_freelist/common_interrupt/do_IRQ/irq_exit/
do_softirq/__do_softirq/net_rx_action/process_backlog/
netif_receive_skb/ip_rcv/ip_rcv_finish/ip_local_deliver/
ip_local_deliver_finish/tcp_v4_rcv/tcp_v4_do_rcv/
tcp_rcv_established/__tcp_push_pending_frames/tcp_transmit_skb/
ip_queue_xmit/ip_output/ip_finish_output/dev_queue_xmit/
__qdisc_run/pfifo_fast_dequeue
```

Figure 13. Maximum stack depth seen (kernel compile).

7. Evaluation

Tralfamadore approaches dynamic analysis in a manner that is dramatically different from previous systems and in which tools may access any point in execution history at any time. This comes with a cost: large amounts of fine-grained trace data must be collected, stored, and processed. In this section, we evaluate the practicality of this approach. First, we measure the overhead of taking and storing raw trace data. We then evaluate the cost of generating and storing trace indices used to speed up queries, and the degree to which they improve the speed and efficiency of various queries. We conclude with a consideration of the class of analyses for which Tralfamadore is currently suited and discuss how it might be evolved to serve a broader set of use cases.

7.1 Overheads of Trace Capture and Storage

Table 2 shows the overheads of trace collection for four different example workloads. *Kernel* is a minimal build of the Linux kernel which balances CPU and I/O, and exercises a variety of kernel services. *OS Boot* traces a virtual machine as it boots. *Postmark* is a standard file system benchmark. Finally, *apache* is a trace of 30 minutes of activity totalling 7981 requests on our departmental web server.

Each workload was run natively, then under virtualization using QEMU 0.9.1 and VMware Workstation 7.0, with and without tracing activated. The host was a 2.5GHz quad-core Xeon server with 16GB of RAM and a 1 TB 7200rpm SATA2 hard drive. Table 2 shows the effects of virtualization and tracing on total workload execution time, as well as the sizes of the traces produced (*Apache* measures request latency rather than total time, since the workload is bursty). QEMU traces are generated using the modifications described in Section 4.2, whereas VMware traces are event logs created using its deterministic logging feature. We include both the

	Native	Qemu	Qemu	Log Size	Log Size	VMware	VMware	Log Size
	Linux	Base	Tracing	(Kernel)	(All)	Base	Tracing	(All)
Kernel	89 sec	1569 sec	7459 sec	101.64 GB	1991.01 GB	87 sec	88 sec	0.002 GB
OS Boot	38 sec	153 sec	549 sec	66.43 GB	128.31 GB	25 sec	31 sec	0.022 GB
Postmark	152 sec	332 sec	523 sec	274.54 GB	297.40 GB	237 sec	310 sec	0.014 GB
Apache	6.17ms	49.98ms	144.34ms	74.63 GB	80.06 GB	10.55ms	26.41ms	0.18 GB

Table 2. Tracing overheads for various workloads.

full size of the trace and the portion of it which represents kernel activity; the analyses we evaluate operate on the kernel component of the trace alone.

The storage requirements and workload performance overhead of direct tracing are high. However, both could be mitigated by using deterministic record-replay to capture execution, and expanding the log into a trace on demand [38]. The VMware columns in Table 2 show the considerable degree by which recording costs could be realistically reduced.

File	Size	Ratio
raw trace	101.6 GB	100%
Invoke index	2.2 GB	2.20%
Context index	324 MB	0.32%
Alloc index	12 MB	0.01%

Table 3. Index size relative to trace size (kernel).

Workload	Query	% trace	% trace
		(large)	(small)
Postmark	Heap Slicing	0.01%	0.00%
	Arguments	0.23%	0.00%
Kernel	Heap Slicing	1.25%	0.02%
	Arguments	0.17%	0.06%
Apache	Heap Slicing	37.5%	0.01%
	Arguments	0.25%	0.00%

Table 4. Amount of trace read when using indices.

7.1.1 Indexing Performance

As shown in Table 2, the full trace grows very rapidly, to the point that scanning it directly can take considerable time. Fortunately, many trace operators only need to see a small amount of the trace, and can make use of indices to read only the relevant sections. Table 3 shows the relative size of the index to the trace for the most used operators in our operator library. As this table demonstrates, the size of the index is generally inversely proportional to the degree of semantic information it provides. This results from the fact that higher-level operators consume filtered information from the operators below them.

The cost of analysis is determined largely by the amount of physical trace that must be read. Table 4 measures the benefit of using indices in terms of the percentage of trace that an analysis must examine when the relevant indices are available (without indices, it would be necessary to read the entire trace). For each of the workloads described at the start of this section, we ran two heap slicing analyses and two argument value extraction queries. The large heap slicing analysis examines every TCP packet passing through the tcp_v4_rcv function, while the small only examines (relatively infrequent) ARP packets passing through arp_rcv. Similarly, the two functions chosen for argument value extraction were the common kmem_cache_alloc, which is involved in most object allocations, and the relatively rare mmap_region. Because the heap slicing analysis must scan the physical trace from the time an object of interest is allocated until it is freed, and because in the Apache workload there is almost always a live TCP packet, the heap slicing query consumed a large amount of physical trace even when indexed. Every other query consumed a tiny fraction of the total trace, completing quickly.

7.1.2 Evaluation Summary

In our current prototype, there is no denying that traces are large and slow to generate. But the premise of Tralfamadore is to decouple trace generation from analysis, and we find that having done so, analysis itself performs well. Because Tralfamadore can aggressively reuse trace data, we are relatively unconcerned with the space required to store it. We also believe that trace-based analysis makes it much easier to develop new analyses, since they can be written in the author's choice of language and with a full complement of libraries, and that being able to freely seek within a trace allows a more natural style of query expression. Furthermore, persistent traces allow users to break the expensive and delicate cycle of hypothesis, instrumentation, execution, and analysis which is generally used for query refinement.

However, it is clear that the ability to collect and analyze traces more easily and with lower overhead would make the techniques described in this paper apply to a broader range of problems. As discussed in Section 4.2, a natural optimization would be to use deterministic logging during the recording phase. The full log could then be treated as a cache that could be regenerated when necessary using replay. We believe that this approach would allow operators to rapidly create the data needed to populate their indices. From this point, some analyses could proceed largely based on the contents of their indices, and use replay to materialize only the relevant regions of the complete trace, on demand. Such an approach could facilitate the processing of much larger amounts of trace.

8. Related Work

As described in Section 2.2, the traces recorded with Tralfamadore are complete, meaning that analyses based on these traces have access to the register and memory state on every instruction boundary. This section compares Tralfamadore with other frameworks that provide access to the same level of information. This includes traditional dynamic binary instrumentation (DBI) frameworks such as Pin [20], and replay based frameworks such as Aftersight [10].

Aftersight [10] proposes to decouple dynamic analysis from execution by using deterministic record-replay to efficiently record the execution of virtual machine and replay this execution in an whole-system emulator (QEMU). Like Tralfamadore, Aftersight supports whole-system or kernel-level offline analyses, but does not provide an instrumentation API, forcing users to directly embed their instrumentation and analysis into QEMU's micro-operations.

SimOS [32] explores the idea of using a machine simulator to study the execution of entire systems. Similar to Tralfamadore, SimOS proposes the use of annotations to capture events at a higher semantic level. As with Tralfamadore, annotations can be recursive (annotations can be built from other annotations).

SimOS can operate in two modes: emulation mode which uses dynamic binary translation and accurate mode which executes the system under a cycle-accurate simulator. Analyses performed in emulation mode execute in an online manner, and do not provide the benefits of offline analysis. Analyses performed in accurate mode may be repeatable if the simulation is deterministic but the overheads in this mode are much higher than with Tralfamadore, ranging between 180 and 6400× for a uniprocessor simulation [31].

Although analyses built with SimOS are composable, SimOS does not address the issue of reusability. Also, SimOS does not support indices to efficiently navigate execution, and therefore does not provide mechanisms such as annotation caching to deal with stateful analyses, as described in Section 4.4.

Traditional binary instrumentation frameworks are fundamentally different than Tralfamadore. These are online tools and are "stuck in time." They are limited to analyzing a program as it executes instead of analyzing an execution. Dynamic binary instrumentation frameworks such as Pin [20] and Valgrind [24] use process-level dynamic binary translation to add instrumentation to a program as it executes. Because both the analysis and the target program execute in the same address space, analysis writers are forced to use clever techniques to avoid clashing with the target program. This technique does not require explicit modifications to the program being analyzed but it is not completely transparent because it may cause memory conflicts between the program being analyzed and the analysis.

These tools are unable to instrument and analyze the execution of operating system code. PinOS [7] addresses this issue by extending Pin with the ability to do whole-system analysis. Because it uses virtualization, most of the instrumentation framework sits outside of the system being analyzed but the instrumentation code and data still need to be embedded within the system, potentially causing memory clashes as with Pin and Valgrind.

JIFL [25] proposes to embed a DBI within the Linux kernel. This approach allows adding arbitrary dynamic instrumentation to the Linux kernel but requires the operating system to be explicitly modified to add support for a just-in-time compiler.

Nirvana [6] is a user-level offline dynamic analysis framework based on a variant of record-replay. Instead of explicitly logging all the external inputs and non-deterministic events, Nirvana captures this information implicitly by logging the complete register state after kernel-to-user transitions and non-deterministic instructions such as `rdtsc`, and logging all values read from memory by the processor while executing.

Nirvana provides an instrumentation API but, unlike Tralfamadore, it does not support capturing and analyzing the execution of operating system code. Nirvana can record and analyze unmodified programs, but it is not completely transparent. Like Pin and Valgrind, the Nirvana execution recorder is embedded in the address space of the program being recorded, which can cause memory conflicts. PinPlay [27] extends Pin to support replay-based offline analysis in a manner similar to Nirvana, and uses a software-simulated cache coherency protocol to deterministically replay the execution of a process on a multi-processor machine. PinPlay provides the same level of transparency as Pin.

Bitblaze [33] is an online whole-system binary analysis framework based on QEMU. Like Tralfamadore and Aftersight, Bitblaze analyzes execution at the processor level which provides a transparent approach to dynamic analysis, and support for the analysis of operating system code. Because Bitblaze is an online framework, it is also "stuck in time" and can only analyze programs as they execute.

Time-travelling virtual machines [15] use deterministic replay to allow a debugger attached to a Linux kernel to step execution backwards as well as forwards. VMware Workstation now includes a record-replay facility [38] that can be used for replay debugging [3] of recorded software. Time-travelling virtual machines support the offline analysis of operating systems but the interface provided to analyze execution is limited to debugging contexts (e.g., breakpoints, watchpoints, etc.)

Omniscient debugging [1, 8, 19, 29] is an approach to debugging where the execution of a program is stored and indexed using techniques borrowed from program slicing [36], allowing debugging to work "backwards in time." All existing implementations of omniscient debugging are limited to user-space programs and provide an interface limited to debugging.

Whole Execution Traces (WET) [39] is a generic trace format with the same completeness properties as traces collected with Tralfamadore. WET traces embed control and data flow information in the trace to provide efficient backward navigation. This information could be added to the Tralfamadore trace format to simplify going backward to get register and memory definitions.

8.1 Deterministic Record-Replay

As discussed in Section 4.2, it is possible to use deterministic virtual machine record-replay to capture a lightweight event log of a running system, which can be reconstituted into a full trace during a later replay phase [38]. Tralfamadore could use this complimentary technique to reduce the overhead on execution incurred by the trace collection facility. The overhead of this approach are very low for single core systems. Recording multicore execution is substantially more challenging due to the need to capture memory races. The overhead can be substantial for workloads that exhibit a large amount of shared memory communication. Fortunately, deterministic record-replay for multicore is an active area of research and recent work both at the software [26, 34], and at the hardware [13, 22, 37] level aim to address this issue. Just as in the single processor case, these emerging techniques will be usable to generate detailed traces of multicore workloads.

8.2 Offline and Whole-System Analyses

TaintBochs [9] uses a mix of online taint tracking and offline trace-based analysis to study the lifetime of sensitive information such as passwords. TaintBochs uses a modified version of the Bochs emulator [17] to record traces that contain all writes to memory and all updates to a specific subset of registers. One could use Tralfamadore to also implement the taint propagation offline, using a recorded execution to experiment with different taint propagation policies.

Dataflow tomography [23] proposes the use of whole-system taint tracking as a tool to understand complex computer systems. This project is complementary to Tralfamadore, meaning that Tralfamadore could be used to implement the various taint policies described in the paper. Using Tralfamadore would have provided the authors with the benefits of deterministic analysis, in addition to using a high-level language instead of dealing directly with QEMU's dynamic binary translation engine.

Introvirt [14] proposes to use virtual machine deterministic replay to retroactively detect the presence of an intrusion. Introvirt is the original source of inspiration for the idea of retroactive assertions described in Section 6.2. While Introvirt checks are expressed at the source level, Tralfamadore retroactive assertions are written as binary analyses and can validate assertions difficult to express at the source level such as detecting a stack overflow.

8.3 Querying Execution

Several projects have explored the notion of querying program execution, either by providing an explicit query interface to the end user or using machine learning to discover execution patterns. Two recent projects from the programming languages community, relational queries over program traces [12], and the program query language PQL [21], have proposed query interfaces to allow developers to search for interesting aspects of program behavior. The scalable omniscient debugging project [29] also explored query interfaces to assist in navigation. A suitable query interface would be an excellent tool for interacting with traced execution. These query languages are designed around specific language runtimes and are not designed to query operating system execution. They operate at a higher level and rely on runtime information provided by the operating system or the language runtime. Tralfamadore analyses execution at the hardware level below the operating system and needs to explicitly reconstruct this information.

9. Conclusions and Future Work

We believe that developers (especially kernel developers) would benefit greatly from being able to explore actual program execution as easily as they can navigate source. To that end, we have built a platform for recording execution of an entire machine as a persistent object, and facilities for easily constructing complex and powerful dynamic analyses from a library of simple, reusable components. We have demonstrated several applications of this system, including an interactive source-level execution browser with interfaces for both control and data flow, and a variety of retroactive assertions against the behavior of an executing system.

The overheads associated with our approach seem high compared to traditional dynamic analysis, but they can be amortized over large numbers of queries. Additionally, trace-based analysis avoids the difficulties of reproducing machine state when performing cyclical analysis, and provides a much richer and more convenient environment for writing interactive dynamic analysis tools. Our evaluation has demonstrated that it is practical to capture and analyze complete traces of kernel activity for reasonable periods of time, including workloads such as kernel compilation, OS boots, and samples of server workloads. Further, once recorded, it is possible to perform highly interactive analyses of these workloads, often receiving complex answers about a control and data flow within a matter of seconds.

Tralfamadore is still in its infancy. While the system already represents a considerable effort, spanning four years and contributions from a group of six graduate students, many challenges remain. We are in the process of releasing a hosted version of the system that allows developers to analyze current and historical versions of the Linux kernel. To address performance challenges and scale, the system is being extended to parallelize and dispatch analysis to a cluster of physical hosts. Once this framework is in place, we hope to explore how Tralfamadore can be used to compare execution across multiple versions of software, which we hope will aid in identifying and diagnosing performance challenges and helping developers to reproduce and identify root-causes for software failures.

Acknowledgments

We thank the VEE reviewers for their insightful feedback and for accepting the paper for publication. We also thank many of our peers, including Paul Barham, Herbert Bos, Steve Gribble, Steven Hand, Rebecca Isaacs, and Timothy Roscoe, for being supportive and encouraging over the course of this work. We acknowledge the graduate students in CS538W 2010 ("Execution Mining") for suffering through project assignments on an early version of the system and helping to find and fix bugs.

As some previous reviewers have pointed out, the Tralfamadore project has involved quite a bit of *engineering*: while many of the techniques used by "Tralf" are well understood, tying together the necessary pieces to perform trace collection, parsing, instruction analysis, as well as source-level interactions through DWARF and symbol files involved a lot more effort than we imagined at the beginning of the project, despite considerable experience in underestimating such things!

References

[1] C/C++ trace-based debugger based on chronicle and eclipse. http://code.google.com/p/chronomancer/.

[2] Vassert programming guide. http://www.vmware.com/pdf/ws65_vassert_programming.pdf.

[3] Replay debugging on linux. http://www.vmware.com/pdf/ws7_replay_linux_technote.pdf.

[4] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *PLDI '90*.

[5] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference*, 2005.

[6] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *VEE '06*.

[7] P. P. Bungale and C.-K. Luk. Pinos: a programmable framework for whole-system dynamic instrumentation. In *Virtual execution environments*, 2007. ISBN 978-1-59593-630-1.

[8] J.-D. Choi, B. P. Miller, and R. H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, 13, 1991. URL http://doi.acm.org/10.1145/115372.115324.

[9] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *USENIX Security Symposium*, 2004. URL http://portal.acm.org/citation.cfm?id=1251375.1251397.

[10] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX Annual Technical Conference*, 2008.

[11] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. In *Operating Systems Design and Implementation*, 2002.

[12] S. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2005.

[13] D. R. Hower and M. D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *International Symposium on Computer Architecture*, 2008. URL http://dx.doi.org/10.1109/ISCA.2008.26.

[14] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Symposium on Operating Systems Principles*, 2005. URL http://doi.acm.org/10.1145/1095810.1095820.

[15] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX Annual Technical Conference*, 2005.

[16] E. Kohler, R. Morris, B. Chen, J. Jannotti, and F. M. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 2000.

[17] K. P. Lawton. Bochs: A portable PC emulator for unix/x. *Linux Journal*. ISSN 1075-3583.

[18] G. Lefebvre, B. Cully, M. J. Feeley, N. C. Hutchinson, and A. Warfield. Tralfamadore: Unifying source code and execution experience (short paper). In *EuroSys*, 2009.

[19] B. Lewis. Debugging backwards in time. In *Workshop on Automated Debugging*, 2003.

[20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, 2005.

[21] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using pql: a program query language. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2005.

[22] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *International Symposium on Computer Architecture*, 2008. URL http://dx.doi.org/10.1109/ISCA.2008.36.

[23] S. Mysore, B. Mazloom, B. Agrawal, and T. Sherwood. Understanding and visualizing full systems with data flow tomography. In *Architectural Support for Programming Languages and Operating Systems*, 2008.

[24] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Programming Language Design and Implementation*, 2007.

[25] M. Olszewski, K. Mierle, A. Czajkowski, and A. D. Brown. JIT instrumentation: a novel approach to dynamically instrument operating systems. In *EuroSys*, 2007. URL `http://doi.acm.org/10.1145/1272996.1273000`.

[26] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 177–192, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: http://doi.acm.org/10.1145/1629575.1629593. URL `http://doi.acm.org/10.1145/1629575.1629593`.

[27] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Code Generation and Optimization*, 2010. URL `http://doi.acm.org/10.1145/1772954.1772958`.

[28] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23–24), 1999.

[29] G. Pothier, E. Tanter, and J. Piquer. Scalable omniscient debugging. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2007.

[30] F. Reiss, K. Stockinger, K. Wu, A. Shoshani, and J. M. Hellerstein. Enabling real-time querying of live and historical stream data. In *Scientific and Statistical Database Management*, 2007.

[31] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: The SimOS approach. *IEEE Parallel and Distributed Technology*, 3, 1995. URL `http://dx.doi.org/10.1109/88.473612`.

[32] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation*, 7, 1997. URL `http://doi.acm.org/10.1145/244804.244807`.

[33] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, N. James, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *International Conference on Information Systems Security*, 2008. doi: http://dx.doi.org/10.1007/978-3-540-89862-7_1. URL `http://dx.doi.org/10.1007/978-3-540-89862-7_1`.

[34] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: parallelizing sequential logging and replay. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 15–26, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0266-1. doi: http://doi.acm.org/10.1145/1950365.1950370. URL `http://doi.acm.org/10.1145/1950365.1950370`.

[35] K. Vonnegut. *Slaughterhouse Five*. Delacorte, 1969. ISBN 0-385-31208-3.

[36] M. Weiser. Program slicing. In *International Conference on Software Engineering*, 1981.

[37] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *International Symposium on Computer Architecture*, 2003.

[38] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Modeling, Benchmarking and Simulation*, 2007.

[39] X. Zhang and R. Gupta. Whole execution traces and their applications. *ACM Transactions on Architecture and Code Optimization*, 2, 2005. URL `http://doi.acm.org/10.1145/1089008.1089012`.