Multi-stage Replay with Crosscut

Jim Chow Dominic Lucchetti Tal Garfinkel
Geoffrey Lefebvre Ryan Gardner Joshua Mason Sam Small
VMware

Peter M. Chen[1]
University of Michigan

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

General Terms Design, Experimentation, Performance, Security

Keywords Design, Experimentation, Performance, Security, Replay, Virtual Machines

Abstract

Deterministic record-replay has many useful applications, ranging from fault tolerance and forensics to reproducing and diagnosing bugs. When choosing a record-replay solution, the system administrator must choose *a priori* how comprehensively to record the execution and at what abstraction level to record it. Unfortunately, these choices may not match well with how the recording is eventually used. A recording may contain too little information to support the end use of replay, or it may contain more sensitive information than is allowed to be shown to the end user of replay. Similarly, fixing the abstraction level at the time of recording often leads to a semantic mismatch with the end use of replay.

This paper describes how to remedy these problems by adding customizable replay stages to create special-purpose logs for the end users of replay. Our system, called Crosscut, allows replay logs to be "sliced" along time and abstraction boundaries. Using this approach, users can create slices that include only the processes, applications, or components of interest, excluding parts that handle sensitive data. Users can also retarget the abstraction level of the replay log to higher-level platforms, such as Perl or Valgrind. Execution can then be augmented with additional analysis code at replay time, without disturbing the replayed components in the slice. Crosscut thus uses replay itself to transform logs into a more efficient, secure, and usable form for replay-based applications.

Our current Crosscut prototype builds on VMware Workstation's record-replay capabilities, and supports a variety of different replay environments. We show how Crosscut can create slices of only the parts of the computation of interest and thereby avoid leaking sensitive information, and we show how to retarget the abstraction level of the log to enable more convenient use during replay debugging.

1. Introduction

Deterministic replay systems provide the ability to record the execution of a computing entity (e.g., a virtual machine, process, or script) and reproduce that execution at a later time. Deterministic replay systems can recreate the complete state of the recorded entity at any point during the recorded execution. The ability enables a wide variety of compelling applications, such as software fault tolerance [2], forensics [5], precise reproduction of crashes, diagnosis and debugging [11, 23, 24], and offloading and parallelization of dynamic program analysis [4].

Deterministic replay systems record an execution efficiently by checkpointing the computing entity, then logging all external data or events that the entity receives. Deterministic replay systems can be built for entities at any level of abstraction, as long as the computing entity at that abstraction level behaves as a deterministic finite-state machine given sufficient recorded inputs. For example, one can record the execution of a virtual machine efficiently by checkpointing the state of the virtual machine, then logging interrupts, incoming network packets, and user input. Or one can record the execution of a process by checkpointing the process's address space, then logging data returned by operating system calls.

Current replay systems require users to choose *a priori* the scope of recording (what entities to record and during which time periods) and at what abstraction level to record it. These decisions affect how much information is captured in the recording and how convenient and efficient it is to replay the recording. Inevitably these decisions make trade-offs with respect to record overhead, replay overhead, and log space.

Making these decisions is easiest if the user knows at the time of recording exactly how the recording will be used during replay. For instance, a programmer who is debugging a program will generally know approximately which processes, time intervals, and abstraction levels may be needed to debug a problem. In these cases, the user can choose the comprehensiveness and abstraction level of the recording solution to best fit the end use of replay.

Unfortunately, in many situations, the user does not know in advance how the recording will be used during replay; rather, the specifics of what is needed during replay are learned later (i.e., at the time of replay). Examples of such situations are recording a computer's execution to enable *post hoc* analysis of intrusions, or recording a suite of communicating applications to provide an execution trace to the developer when one of these applications crashes. In these situations, the user cannot make an optimal choice at the time of recording as to when, what, and at what abstraction level to record, and must instead record so as to enable a broad range of potential uses.

For example, consider the question of what abstraction level to record. If the end use of replay is not known, fixing the abstraction level at the time of recording may lead to a semantic mismatch

[1] Peter Chen was supported by NSF award CNS-0614985.

VEE'10, March 17–19, 2010, Pittsburgh, Pennsylvania, USA.
Copyright © 2010 ACM 978-1-60558-910-7/10/03. . . $10.00

during replay. E.g., if the system records at the level of machine instructions, it is difficult to use the log to conduct replay debugging at the level of interpreted languages [10]. On the other hand, if the system records at the level of an interpreted language, it may miss important behavior at lower levels of abstraction. Simultaneously recording at multiple levels of abstractions allows a variety of uses later but increases overhead during recording.

Similarly, consider the question of which entities to record (say, among several processes). A simple strategy is to record each entity separately, then replay the set of entities once the requirements for replay are known. A better strategy is to record the entities as a group, since this can reduce storage and time overhead during recording by avoiding the need to record communication between the entities. However, if the end use of replay is not known, the user cannot pre-determine which group to record. The user could choose to record all processes individually, but this increases recording overhead. The user could also record all processes as a single set, but this increases replay overhead, which can be particularly problematic when replaying multiple times (for example when replay debugging emulates a reverse breakpoint with repeated forward replay steps[11]).

Additionally, consider the question of when to take a checkpoint during recording. Since replay always starts from a checkpoint, having more checkpoints allows an end user of replay to more quickly replay to the point of interest. Checkpoints also allow end users of replay to skip over an interval of execution, and this can be used to elide confidential information from a recording (which may need to be done before sharing a recording with developers or analysts). If the end use of replay is not known, the system must guess when to take a checkpoint. Guessing incorrectly make the resulting recording less convenient or less useful during replay. To preserve ultimate flexibility during replay, the system should take checkpoints frequently during recording, but this greatly increasing time and space overhead during recording.

These problems are caused by the two-stage (log and replay) design of today's record-replay solutions, in which the final replay stage takes as input the log that was captured originally. This forces the end use of replay to match the logging system in the scope being replayed and the abstraction level at which replay takes place.

Our goal is to provide a recording and replay system that can record and replay efficiently, even if the end use of replay is not known until after the recording. Our insight is that, while the comprehensiveness and abstraction level of *recording* must be fixed *a priori*, the comprehensiveness and abstraction level of *replay* can be determined later, i.e. when it is better known what information and abstraction levels are required during replay.

We present a system called Crosscut that leverages this insight by taking a multi-stage approach. Crosscut first records the execution comprehensively and efficiently at the level of machine instructions. Comprehensive replay systems are attractive because they can record the state of an entire system (e.g., an entire virtual machine [2, 5]), which enables analysis of all activity on the system. Further, because they only need to record non-deterministic inputs that enter the system from the external world, recording overheads can be remarkably low (often under 5% [28]). Crosscut thus preserves all information until it is known which information is needed during replay.

Later, once the end use of replay is known, Crosscut transforms the log before it is shipped to the end user of replay. This transformation takes advantage of the user knowledge that exists at the time of replay but that did not exist while recording, such as what processes, files, and semantic levels need to be examined. The transformations between the initial recording step and the final replaying step allow Crosscut to avoid the problems typically encountered by comprehensive record-replay systems, such as including sensitive information and semantic mismatch with the end user of replay. A transformation step accepts as input one log and generates a new log that is optimized toward a particular use of replay.

Crosscut can transform a log in several ways. First, Crosscut can selectively *slice* out portions of execution that are not of interest, such as time periods, applications, or components not being analyzed. Crosscut can slice out execution contexts that handle sensitive data, helping to mitigate the privacy concerns associated with sharing and storing replay logs. Crosscut can also slice the log along abstraction boundaries, such as the kernel/user boundary, or interpreter/program boundary, or even individual module boundaries, resulting in a log that can be replayed on a higher-level replay system, thus more easily facilitating high level analysis (e.g. leak and memory error detection) and debugging.

Crosscut transformation steps are implemented using a novel technique called *relogging*, where execution is replayed using one log and filters are applied during replay to generate a new log. Thus, Crosscut uses replay itself to transform a recording into a more efficient, secure, and usable form for later downstream uses of replay.

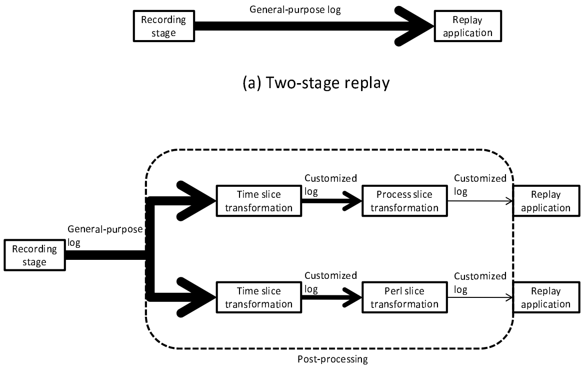
Our work makes three main contributions. First, we develop the idea of multi-stage replay, which generalizes the two-stage design used by prior replay systems. Second, we show how to use *relogging* to take a recording and create a new recording that is customized for a particular abstraction level, scope, and time interval. Third, we demonstrate multi-stage replay by implementing two basic transformations (time slicing and abstraction slicing), and we compose these basic transformations to create a variety of customized recordings, such as a pruned snapshot and log, a subset of processes that can be replayed without OS support, a Perl-level recording, a omniscient debugging log, and a log in which sensitive information has been redacted.

The next section explains how deterministic replay systems record and replay an executing system, and we describe how to generalize the replay workflow by transforming the log via relogging stages. Section 3 describes how a recorded computation can be graphed in two dimensions of time and abstract level and how we can create slices along these dimensions to create meaningful subsets of the computation. Section 4 presents our current implementation, exploring how slicing, recording and replaying are supported at various abstraction levels (virtual machine, process and scripting language). Section 5 evaluates how well Crosscut meets our goals of reducing the size of a recorded computation, speeding up replay, supporting replay at higher levels of abstraction, and ensuring privacy. Section 6 describes related work, and Section 7 concludes.

2. Multi-stage replay

Deterministic replay systems reproduce a machine's computation by leveraging the fact that if a machine is deterministic, and one can record its initial state and all non-deterministic inputs, then one can exactly reproduce the machine's execution, state by state. This method applies to machines at many different levels, such as a physical processor [27], a virtual machine [2, 5], a JVM [16], a process [23], or a Perl interpreter.

Current replay systems work in two stages (Figure 1a). The first stage captures a recording of a system's computation, and the second stage replays the captured recording to regenerate the computation. For recording, the initial state of the machine is captured before execution starts. Once execution starts, all inputs to the machine are recorded along with timing information that specifies the exact point at which the input arrived. Replay works by restoring the machine from the captured snapshot and restarting the computation. When the computation reaches a state that matches the timing information stored in the log, inputs from the recorded run are sup-



(a) Two-stage replay

(b) Multi-stage replay

Figure 1. Two-stage and multi-stage replay. Thicker lines denote larger logs.

plied to the computation. The result is that the replayed execution proceeds exactly as it did the first time.

Limitations of two-stage replay While the simplicity of this design is appealing, it suffers from two major weaknesses. The first weakness is that a two-stage replay system uses a single, general-purpose log, rather than one that is customized to any particular use of replay. This single log works well only when the designers of the system know *before the capture stage runs* what information will be needed during replay, such as what time period and execution features are of interest to the user of the replay system. With such knowledge, the designers can tailor the capture stage to save only the information that will be needed during replay.

However, if the designers do not know, before the capture stage runs, what information will be needed during replay, the single, general-purpose log forces a designer to choose between sufficiency and efficiency. If the designers want to guarantee sufficiency, they can err on the side of capturing more information than might be needed. This extra information increases the size of the checkpoint and replay log, slows replay, and leaks more sensitive information from the logged system to the replaying site. On the other hand, if the capture stage errs on the side of being more efficient, it risks not capturing the information that is needed during replay.

In several important uses of deterministic replay, it is difficult or impossible to anticipate beforehand what information will be needed during replay. For example, when using replay for computer forensics, it is not generally known before the intrusion what information will be needed to analyze the intrusion, determine how the attacker broke into the system, or determine what damage occurred during the intrusion. Similarly, when using replay for debugging transient or non-deterministic bugs, it is not generally known what information will be needed to track down the bug.

The second weakness of a two-stage replay system is that all the work of generating the recording takes place during the initial capture stage, and this work may add significant overhead to the system being recorded. If this overhead is high enough, it may render the recorded system unusable, or it may slow the recorded system so much that the recorded computation is no longer realistic. For example, the Chronicle tool for Valgrind, which records the detailed execution of a program, slows a program by 100-300x [12, 20].

Adding transformations To fix these weaknesses, we add processing stages between the capture stage and the final replay stage (Figure 1b). Each intermediate processing stage transforms a recording in some way, tailoring it for the final use of replay.

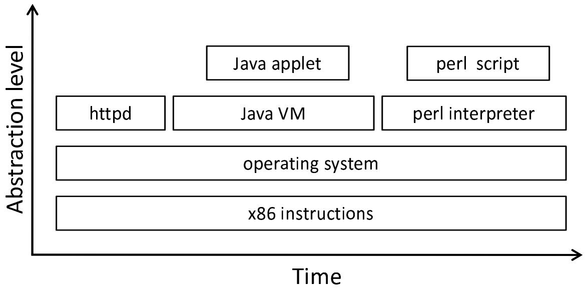


Figure 2. Two-dimensional graph of a recorded computation.

Adding processing stages between the capture stage and final replay stages benefits the system in two ways. First, it resolves the conflict between efficiency and comprehensiveness. The initial capture stage can record the computation as comprehensively as possible. Later, when it is known what information will be needed during the final use of replay, processing stages can then tailor the recording appropriately. The tailored recording can be much smaller, faster and more flexible to replay. E.g., in our evaluation (Section 5.1), Crosscut reduced the snapshot size for a server workload from 4 GB to 4 MB (which reduces the time needed to ship to a remote site), and Crosscut reduced the replay time for IIS from 226 seconds to 19 seconds. The tailored recording also leaks less sensitive information to the replay site. E.g., Crosscut can remove all or part of the execution of processes that handle passwords.

Second, tailoring the recording after the initial capture stage can often reduce the overhead of the initial capture stage. The work to tailor the recording occurs after the initial capture stage and can be done at a later convenient time, e.g., when the system is idle. This decoupling of work works especially well when it is possible to capture an initial comprehensive recording with little overhead, as is the case with virtual-machine recording [4]. For example, deferring the use of Chronicle until replay can reduce the overhead during recording by 100-300x.

With multi-stage replay, the goal of each processing stage is to tailor the recording for a specific end use of replay. The input to a stage is a recording of some computation. The output of a stage is a new recording, either of the same computation or of a subset of the original computation.

We implement each processing stage using a technique we call *relogging*. The idea in relogging is to structure each processing stage as a replay of the prior recording. As the prior recording is being replayed, the processing stage captures or generates information for the new recording. The next section describes the types of transformations that can be done in a processing stage and how a stage generates or captures a new recording during relogging.

3. Replay transformations

This section describes two types of transformations that can be performed on a recorded computation: time slicing and abstraction slicing. The goal of these transformations is to create a new recording that is smaller and faster to replay than the original one, leaks less sensitive information, and can be used more flexibly.

To understand the two types of transformations, it is helpful to graph the recorded computation as a layered stack of running abstract machines (Figure 2). The X axis of the graph represents time. The Y axis of the graph represents the level of abstraction: higher Y values in the graph represent the execution of a higher-level abstract machine. For example, the lowest level could be *x*86 instructions. The next level up could be user-level processes, which supplement the *x*86 instruction set with system calls to the operating system kernel. The next higher abstract machine could be a Java VM, which provides the abstraction of Java bytecode instructions. Still higher levels can also be imagined, such as a Java program that implements an SQL or Perl interpreter, or a Python program that implements a Web server, which executes HTTP requests.

The two types of transformations we use each create a subset, or *slice*, of a recorded computation along one of its two dimensions. A slice is a new, standalone recording of some abstract machine, capturing a continuous portion of the execution history of that abstract machine. Like all recordings, a slice requires an initial snapshot and a log of inputs. We generate the snapshot and input log for the new slice of execution by relogging a prior recording (which could itself be a slice).

3.1 Time slicing

The first type of transformation, *time slicing*, extracts the time intervals of interest from a recording. This transformation yields a variety of potential benefits. The starting snapshot can be substantially smaller, making it easier to store and transport. Restoring snapshots can be faster because a snapshot is smaller, and replay can be faster because the time slice can be customized to only include the time period of interest. By reducing the size of the snapshot and replay log, time slicing can reduce the amount of sensitive information leaked by replay. To meet specific privacy goals, additional analyses can be applied to slice out intervals that handle sensitive data.

Time slicing works using a simple twist on replay. As usual with deterministic replay, the system replays a slice by starting from a snapshot and executing forward using the portion of the log from that slice. The twist comes by noting that because a slice contains a finite, known interval of execution we can determine exactly which state (e.g. disk blocks, memory pages) is accessed during the interval of interest, and include only that state in the snapshot. In contrast, the snapshot for the original, comprehensive log must include all state because it cannot pre-determine (when it takes the snapshot) what state will be accessed in the future.

A computation of interest may be spread over several slices, and treating each as a separate, standalone recording is cumbersome. The initial snapshot of each slice can represent a large amount of state, and in many cases, much of this state will be redundant with nearby slices.

These problems can be alleviated by *stitching* the slices together to form a single, coherent recording. Instead of having separate initial snapshots for each slice, we can encode the snapshot of each slice relative to the ending state of the previous slice. In other words, by applying the difference between states, we can get from the ending state of one slice to the starting state of the next slice.

By recording these differences in a log entry that joins the logs of the two slices, we can create a single recording, with one initial snapshot, and a set of log entries that represents a coherent but discontiguous computation.

3.2 Abstraction slicing

The second type of transformation, *abstraction slicing*, transforms the level of a recording so that it describes the execution of a higher-level abstract machine. For example, a VM-to-process filter would transform a recording at the level of a virtual machine into a recording at the level of an operating system process. Whereas the original recording would describe the state of the virtual machine and log the non-deterministic inputs to the virtual machine, the new recording would describe only the state of the process and the inputs to that process.

Abstraction slicing reduces the comprehensive of a recorded computation in two ways. First, it eliminates from the computation all state and input that pertain to lower abstract machines. For example, the VM-to-process filter would eliminate all inputs to the virtual devices; instead, it would include only inputs to the operating system process.

Second, abstraction slicing will usually include only a subset of the peer entities running at the level of a given abstract machine. For example, a VM-to-process filter will usually retain the history of only some operating system processes.

Reducing the comprehensiveness of a recorded computation is helpful in several ways. Perhaps the most important benefit is improving the flexibility of replay. Because lower-level abstract machines are not included in the transformed history, these levels can be modified during replay without perturbing the replay of the higher-level abstract machine. For example, a VM-level history forces the replay system to replay the entire virtual machine exactly as before, and this makes it difficult to perform tasks such as attaching a debugger or running in-system analysis tools (e.g., Valgrind). In contrast, a process-level history requires the replay system to replay only the process, and this allows a user of replay to attach a debugger or use other tools, as long as those tools do not perturb the execution history of that process.

In addition, abstraction slicing reduces the size of the initial snapshot, since the states of the lower-level abstract machines and other peer entities are no longer needed. It may also reduce the size of the input log, since some non-deterministic inputs to the lower-level abstract machine may not result in non-deterministic input to the higher-level abstract machine. [1]

Third, abstraction slicing reduces replay time. Restoring the snapshot is faster because the snapshot is smaller, and replay is faster because only the higher-level abstract machine needs to be replayed.

Finally, abstraction slicing can reduce the amount of sensitive information contained in an execution history, since it contains state only of the higher-level abstraction machines of interest.

Generating an abstraction slice requires a system that understands the higher-level abstract machine in two ways. First, the system must understand the state representation of the higher-level abstract machine, since this is required to take a snapshot of the machine. Second, the system must understand which inputs are non-deterministic with respect to the higher-level abstract machine.

Extracting the state and non-deterministic inputs of an abstract machine can be implemented in two ways. The first way (*reflection*) is to enlist the help of the layer that implements the higher-level abstract machine (e.g., a VMM implements a virtual machine or an operating system that implements a user-level process). We call the layer that implements the higher-level abstract machine its *interpreter*. Reflection is the easiest way to extract the state and non-deterministic inputs of a higher-level abstract machine. For example, since the operating system implements the abstraction of user-level processes, the operating system can easily extract the state of a process and log all non-deterministic inputs to that process. We use reflection to generate a slice at the level of a Perl script.

The disadvantage to using reflection is that all code to extract the state and non-deterministic input must already be present and active in the original execution history of the interpreter. This is because

[1] On the other hand, it may increase the size of the input log, since input data that is non-deterministic with respect to the higher-level abstract machine may be deterministic with respect to the lower-level abstract machine. For example, the data returned by read system calls is non-deterministic from the point of view of a single operating system process (since other processes may change the file), but it may be deterministic from the point of view of the entire machine.

abstraction slicing is implemented via relogging, which involves replaying the lower-level abstract machine (including the interpreter) exactly as before. When we use reflection to generate an abstraction slice, we discard the extracted state and non-deterministic input during the original run, but save it during the relogging phase.

The second way (*introspection*) to extract the state and non-deterministic input of an abstract machine is by adding code outside the domain of the lower-level abstract machine [6, 10]. For example, code running outside a virtual machine can peek inside the running virtual machine and extract information about that virtual machine without perturbing its state. Introspection allows one to add the functionality to extract state and non-deterministic input after the original execution is logged. However, introspection can be cumbersome to implement and slow to run. We use introspection to generate a slice at the level of a operating system process.

4. Implementation

Our current Crosscut prototype is built on the VMware record-replay stack, which enables replay for x86-based uniprocessors. Using Crosscut begins by using the VMware VMM to capture a virtual machine-level recording [28]. Crosscut can then generate a new recording containing only the time interval, individual OS processes, or Perl scripts of interests. Using pattern matching or dynamic taint analysis, it can remove portions of execution time and state that might reveal sensitive data. Once we have sliced our log appropriately, it can be replayed in a variety of replay environments, including modified versions of Valgrind, VMware, or Perl.

While our current implementation supports a limited set of slicing policies, extending it is straightforward. The infrastructure for slicing Perl is directly applicable to other languages such Python or Java. For example, one can extract the activity of particular functions, modules, or components simply by specifying to Crosscut which time quanta to preserve.

4.1 Virtual machine record and replay

Crosscut supports whole-machine replay: the instruction-for-instruction replay of all *x*86 machine instructions executed on a computer. The state of a whole-computer's computation is captured completely by the runtime state of the CPU, memory, and devices connected to the machine. Non-deterministic inputs are those that cause the CPU or devices to behave in a way not determined by the machine instructions that are replaying. The arrival of a packet on the wire to a network card or the delivery of an interrupt on the CPU are two examples of non-deterministic input.

Crosscut uses virtual machines to record the operation of a whole computer [2, 5, 11, 28], building on the replay support in the VMware VMM [28]. VM replay systems are highly efficient in time and space [5, 28]. A study [28] of VMware's replay implementation showed recording overheads as low as 0.7% and an average of 5% for SPEC benchmarks. Other implementations have shown similar performance [5] with logging rates on the order of KB/s.

The replay of the *x*86 computer as an abstract machine is a basic capability in Crosscut and an important one—though abstraction slicing can generate new recordings at many different abstraction levels from a single recording, it can only generate recordings for abstraction levels above or equal to the original source material. It cannot faithfully generate recordings for abstraction levels below the base level; that data is lost. Consequently, replay systems that operate at lower levels guarantees a broader level of software support with a correspondingly greater capacity to generate new recordings of higher-level abstractions with diminishing need to anticipate what needs recording (that is, reducing the need to predict and possibly over-or-under estimate the need for enabling recording).

Supporting all machine instructions in a computer is the lowest level of software recording we can support, and therefore provides the greatest capacity for doing abstraction slices with the least anticipatory need. In other words, all software on a computer can be recorded, from device driver to Javascript, and all the information needed for faithful new recordings of *any* higher-level abstract machines exists in a single machine-level recording.

4.2 Process slicing

Crosscut allows a collection of processes to be sliced from a recording and later replayed in either a modified virtual machine monitor (VMware workstation), or process level replay environment (Valgrind). Process slicing combines ideas from abstraction slicing and time slicing: it raises the abstraction level from a virtual machine to an operating system process, and it eliminates periods of execution during which processes of interest are not executing.

We begin by looking at how processes are sliced out of a VMM, then discuss replay support.

4.2.1 Process recording

Crosscut represents the computation of an individual OS process with two notions: process checkpoints and state injections. An important aspect of these abstractions is that they are independent of the specific OS or OS version being recorded. Crosscut maintains this property by focusing on the process and the hardware: it is the process's execution on the hardware that is replayed, not the underlying OS it ran on. This allows Crosscut to replay a recording of a process running in one OS on top of a different OS (Windows on Linux, for example).

Process checkpoints Process checkpoints fill the role of the initial snapshot of computation needed by recordings of an abstract machine as described in Section 2. Crosscut's process checkpoints are similar to traditional process checkpoints [13] but differ in an important way: Crosscut's process checkpoints don't include OS state for the process's used resources, such as open file descriptors or sockets; they only include state accessed by the process itself.

Process checkpoints store the contents of the process's user-addressable memory and the contents of user-visible state in the CPU (for example, the instruction pointer and general purpose registers) at an instant in the process's computation. This is similar to the state contained in a process's core dump.

However, an important part of the runtime state needed in process checkpoints isn't included in traditional core dumps: the mapping of virtual addresses for the process. The mappings must be included in the compute state for the process because, though they aren't directly observed or manipulated by the process (which doesn't have access to the tables directing the hardware MMU), the process can observe their effect: physical pages that are mapped multiply times in the virtual address space effectively mirror modifications done to any single page to all pages in the set. Trying to emulate this behavior per operation at replay time without mapping information would be costly.

State injections Crosscut props up the replaying execution of a process whenever the process expects to receive state from the outside: although the checkpoint will include mapped memory at the start of computation, during the course of execution, a process can swap or map more memory into its address space, it can make calls to libraries not in its address space (that is, make system calls to the OS), and it can read write-shared memory directly mapped and written to by other processes.

Crosscut replays these events through *state injections*, which are based on the observation that all sources of non-determinism visible to the process can be summarized by their effect on its compute state: the contents of addressable memory, registers, or memory mappings. State injections store these modifications: the results of system calls, pages swapped or mapped in, and modifications to write-shared memory by other processes are simply recorded as data so they can be supplied later on during replay.

An important property of state injections is that they can be completely represented as a manipulation of the process's runtime state: they do not depend on the OS or the OS version the process runs on. The replaying system need not understand anything of the semantics of paging or system calls, only that the result of one of these operations was that some state was injected at a particular time.

Slicing a process All of the state for creating a process checkpoint and the state injections for a process exists in the recorded whole machine execution. The VMM extracts this information from the recorded computation via introspection to generate a process-level recording.

Recording can begin at any point of a process's lifetime at which it is actually executing on the CPU. Process recording begins with creating a process checkpoint by walking the hardware page tables of the process: Crosscut saves the contents of user-addressable memory, the address space mapping specified by the page tables, and the CPU state of the process.

Although recording can begin at any instruction boundary within the process, some instruction boundaries are more interesting than others. Crosscut can identify particularly interesting events like process creation and destruction with the help of a driver added to the guest (on Windows) or a modified kernel (on Linux).

State injections occur as the result of some OS-specific activity, such as responding to a system call, or swapping in a page. A key observation allows Crosscut to detect them without hooking the OS: the process can only observe these modifications when it runs, and these external modifications to process state only occur while the process is not running on the CPU. This allows Crosscut to boil the execution of the process down to periods where it runs on the CPU and periods where it isn't running on the CPU. In between these periods, state may be injected.

When the process runs on the CPU, no injection events occur. When control flow exits the process, Crosscut detects injected memory state by trapping on writes to the process's address mappings (its hardware page tables) and writes to its addressable memory contents, and observing changes to the registers on entry back into the process. The VMM accumulates these writes and outputs them to the log when control returns back to the process. The VMM inserts itself on every hardware fault, interrupt, and system call path to detect all possible exits and entries from the process.

4.2.2 Process replay

Crosscut currently supports targeting sliced processes to two different execution environments, a process level replay environment in the form of Valgrind and a whole system replay environment, in the form of VMware. Each environment has its own benefits. Valgrind supports a wide range of dynamic program analysis and debugging tools at some cost in performance. The VMware VMM is a less friendly environment for developing fine-grained dynamic instrumentation, but it provides excellent performance and a replay debugging environment with a variety of unique capabilities. Both support replaying Windows and Linux processes.

Replay in Valgrind Valgrind [18] is an open-source dynamic binary instrumentation tool that executes compiled binaries on Linux and dynamically instruments them for the purpose of runtime analysis. In Crosscut, Valgrind is a execution target for replaying process recordings. An important property of Crosscut process recordings is that they are independent of OS semantics, and this is taken advantage of in our Valgrind replay environment: it can replay recordings of Windows or Linux processes even though Valgrind itself only runs on Linux and is only designed to run Linux programs.

One of our goals is to retain Valgrind's ability to add instrumentation and runtime analysis to running (for us, replaying) programs, so long as the instrumentation doesn't mutate the computation of the process (a design goal for Valgrind tools [18]).

Because Valgrind knows only how to execute programs, not replay them, we modified the normal workings of Valgrind in three ways: constructing address spaces, branch counting, and event handling.

Replaying a process recording bypasses Valgrind's usual startup sequence, which implements its own program binary loader. The modified sequence recreates the recorded process's address space on startup by `mmap`'ing the pages from the process checkpoint at appropriate places in the process address space. A caveat to this is that Valgrind and its binary translation code cache also live in the process's address space, which raises the possibility of collision. However, this problem already exists in Valgrind's normal operation [18], and Valgrind's usual solution to this also applies: allow the user to recompile Valgrind to occupy some other free spot in the address space (about 2.5MB) if the carefully chosen default location doesn't work.

Another issue to recreating the address space occurs with recorded Linux processes. One Linux user-level page cannot be recreated because even an otherwise empty address space has a non-overwritable copy of the page: the `vsyscall` page, which is a shared page the kernel maps into the process to accelerate some system calls. One option is to modify the kernel on the replaying host to allow the user to supply their own copy of this page, but we have found that the page and its entry points don't change enough that the host's copy needs to be replaced. Moreover, collisions with the `vsyscall` page aren't a problem with Windows programs replaying in Valgrind, because 32-bit Windows programs don't use address space above the 2GB mark where the `vsyscall` page lives.

State injections happen at specific points in the replayed instruction stream. Process recordings use a tuple of branch counts, instruction pointer, and loop counter to identify these points, a strategy that has been described by other replay systems [5]. Performance counters won't suffice to provide branch counts in Valgrind, since dynamic instrumentation, which we want to allow to augment replaying execution, can add an arbitrary amount of branches to the original execution. Instead, our system counts branches in software [14]. When branches are translated, the code cache contains a small translation for branch instructions that increments a software branch counter.

Each branch translation also checks the next event to see if a state injection should be delivered. When it is, Valgrind replays a modification to the process's compute state: either program memory, the address space mappings, or the CPU. Because state injections completely encapsulate the process's interaction with the OS, Valgrind never has to execute an actual system call on behalf of the process. This allows a process to be replayed on a different machine, with a potentially different OS.

Replay in VMware Crosscut can also replay process recordings in the VMware VMM. However, the state for replaying a VM doesn't exist in a process recording. To fix this, Crosscut uses a "shell" VM state to provide a environment for replaying the process recording. The initial VM state provides basic initialization of the CPU: for example, putting it into protected mode, or turning on paging (currently Crosscut uses a normal VM checkpoint for this purpose, included when the process recording is created, but without the full contents of machine memory). The stub initialization provides an environment for loading the process checkpoint into VM memory: process pages load into machine memory, the mappings of the process are written into hardware page tables, and the user-visible portions of the CPU are loaded. At this point the VM is ready to run.

Replaying the process is just a matter of allowing it to run, and delivering state injections at appropriate times. These injections use VM replay delivery mechanisms based on counting of performance events to trigger delivery. When triggered, the VMM reads data from the recording and modifies the VM state accordingly, by writing to the page contents, the page tables, and the CPU.

An important aspect of replaying the process is that most state normally associated with whole VM execution doesn't exist while replaying process recordings. For example, the interrupt vector table and the OS memory are neither present nor needed—state injections by the VMM take the place of all faults, interrupts, or system calls that the process relies on. Execution never depends on these hardware events occurring during guest execution. Thus, the VMM supplants the OS's role when running the process.

4.3 Perl slicing

Crosscut's Perl slicer accepts as input a recorded computation of an entire virtual machine and produces the recorded computation of a Perl script that was running in that virtual machine. The resulting Perl slice can be replayed apart from the original virtual machine. This makes the Perl slice much easier to use in debugging than the original recording, because the execution of the rest of the system can be modified without affecting replay. In particular, the Perl interpreter's execution can vary from the original run, and this allows the programmer to use the debugging features of the interpreter, such as breakpoints.

As always, a recorded computation or slice consists of a checkpoint and a log of non-deterministic inputs. To simplify the implementation, Perl slices produced by Crosscut always start at the invocation of a script. This restriction allows Crosscut to use the Perl interpreter's script-invocation code as its method for restoring from a checkpoint, so the script program file functions as the checkpoint.

Crosscut captures the log of non-deterministic inputs by relogging the original virtual machine recording and using reflection to extract the state and non-deterministic inputs for the Perl script. The non-deterministic inputs to the script are represented as inputs into the Perl interpreter, which cause it to recreate the script-level entities (script variables and values) that allow the script to progress. External inputs to a script come from library calls that are implemented by the Perl runtime. A property of this is that the script recordings never refer to script-level entities or their representation in memory themselves. Instead, these entities are regenerated (in whatever representation the script's interpreter desires) as the interpreter replays its execution based on the recorded input. Crosscut records and replays input to the interpreter at the system call level, such as the results of `read` or `stat`.

Crosscut must distinguish between two sources of system calls. System calls that are caused by the script are part of the recorded computation and must be recorded or replayed. System calls that are caused by other actions of the Perl interpreter are not part of the recorded computation, may vary during replay (e.g., due to debugging operations), and must *not* be recorded or replayed. This allows Crosscut to record and replay the execution of the Perl *script* without recording or replaying the Perl *interpreter*, which allows the script to be replayed on an interpreter whose execution varies due to debugging, or on a different interpreter instance entirely.

To distinguish between these sources of system calls, we added a *constrain bit* to the Perl interpreter. It remains on as long as the script is executing, i.e. the Perl interpreter is executing a script instruction. It is turned off while performing other tasks in the interpreter's execution. Only system calls executed while the bit is enabled are recorded.

In addition to system calls, the interpreter records signals that are delivered to the script. The interpreter identifies asynchronous delivery points for these signals with the addition of a Perl instruction counter to the interpreter.

Replay forking For deterministic replay to work, the instructions and state of a replaying virtual machine must match exactly those executed during the original run. This requirement would be violated if we ran the reflection code inside the replaying virtual machine during relogging but not during the original run. We considered two ways to meet this requirement during Crosscut's relogging step.

The first way is to embed and run the reflection code in the Perl interpreter during the original run. During the original run, the captured data can be discarded, since it can regenerated and preserved during relogging if needed. Although this idea is straightforward, it may degrade performance during the original run.

To avoid degrading performance, we do not run the reflection code during recording; the code is present but not active. During relogging, the reflection code is activated and captures the necessary data, then the changes made to the virtual machine's state by the reflection code are rolled back, and execution continues. We call this approach *replay forking*.

An important aspect of replay forking is that it leverages code already compiled and contained within the guest VM. This code can directly access higher-level abstractions (e.g., program-level variables), which otherwise would need to be reconstructed by the VMM at considerable effort.

There are three important parts to replay forking: *in situ*, deviceless VM fork, and triggering. Replay forking in Crosscut is done *in situ*: the state and thread of execution of the parent VM is temporarily reused by the child to go along a new fork of execution. No new VM state (including devices) is created for the child, except that memory written by the child is stored in temporary copy-on-write buffers. The child is prevented from modifying devices or much of the privileged CPU state (such as disabling protected mode); it produces output only through *hypercalls* (system calls that trap directly to the virtual machine monitor). The fact that so little VM state can be modified by the child means that reverting back to the parent VM is substantially simpler than a full VM checkpoint/restore, and is a extremely fast process: COW memory is thrown away, the CPU is pointed back to the fork point, and the parent VM continues. A whole roundtrip of VM fork, child execution, and returning to the parent takes only 100 microseconds on our machines.

Replay forks are triggered by a few extra machine instructions, which are added during compilation at sites where reflection code needs to run. These instructions are simple memory operations and branches and have minimal impact during recording. During replay, the VMM sets the page table protections so the MMU traps on these instructions, then the VMM initiates a replay fork. We explored other ways to trigger the replay fork, such as using hypercalls. However, these other methods incur trap and context switch overheads during *recording* and relogging, whereas our mechanism incurs these overheads only during *relogging*.

Perl replaying The execution target for replaying extracted Perl recordings is the Perl interpreter. The same interpreter used in the VM during recording is used outside the VM to replay the generated script recording. The fresh interpreter is pointed to a copy of the Perl program to run, and inputs are fed in from the recorded log.

The same constrain bit that directed the saving of the interpreter's system calls also directs the loading of these values from the recording. Because the constrain bit constrains only certain system calls done by the interpreter on behalf of the script to be fixed by the log, the rest of the interpreter's execution is fresh, and is free to change. For example, we can use the running interpreter's debugging facilities to break the replaying execution of the script and inspect variables and values, all of which occurs newly during replay without being present in the recording.

4.4 Time slicing and stitching

Crosscut implements time slicing to carve periods of execution out of a recorded computation. Crosscut can then stitch together several time slices into a single recording.

Time slicing in Crosscut is implemented at the level of operating system processes. To create a time slice, Crosscut must capture a snapshot at the beginning of the time slice, then capture the non-deterministic inputs during the interval. Both these tasks leverage the framework implemented for process slicing. Crosscut captures the snapshot by replaying the computation until the beginning of the time slice, then saving the state of the process as described in Section 4.2. Crosscut captures the non-deterministic inputs by logging the state injections issued during the target interval.

To stitch two time slices together, we must transform the state at the end of the earlier time slice into the state at the beginning of the later time slice. The simplest way to do this is to store the entire state for the snapshot of the later time slice, but this method stores much more state than necessary because most of the state is likely the same (e.g., the disk is unlikely to change much between slices). Instead, while relogging, Crosscut computes and stores the difference between these states. The current Crosscut prototype can compute the difference between two states at page or byte granularities. To compute the difference at the granularity of pages, Crosscut uses page protections to track the set of pages modified from the end of one slice to the beginning of the next. To compute the difference at the granularity of bytes, Crosscut computes the byte-by-byte exclusive-or on the pages modified between the end of the first slice and the beginning state of the second slice, the compresses the difference using run-length encoding.

4.4.1 Slicing for privacy

We use time slicing to implement two information-redacting policies in our current version of Crosscut. One, substring redaction, is a fully automatic "best-effort" approach similar to current data leak prevention tools. The other, taint redaction, is a semi-automatic approach, whereby developers can annotate their programs to indicate which data is private, and these annotations will subsequently be used by a dynamic taint analysis to infer which execution intervals to cut from the log.

Substring redaction With substring redaction, we want to eliminate periods of time where a particular string exists in memory. For example, we can eliminate the processing of a private key or password from the recorded execution of a login handler, such as an SSH daemon. Doing so can preserve some interesting computation that is useful to replay, but eliminate aspects of the computation that are sensitive.

Substring redaction monitors both the entry of sensitive information (from an external input) and its exit (when the value is gone) while replaying and re-generating a log. When a string of interest is found within the external state being delivered to a recorded computation (such as the result of a system call in a process slice), subsequent computation is removed through a time slice. The slice ends when an inspection of memory reveals the string no longer appears (performed when the system reaches a replay event). The finished time slices are stitched together as they are generated to form a complete recording.

Taint redaction Crosscut also implements redaction based on taint analysis [17]. Taint policy is driven by developers, who identify sensitive data in their programs by tainting them with a provided API. Crosscut ensures that generated process recordings will not have have the tainted data present.

Crosscut splits taint redaction into a multi-stage process: it first extracts a process, identifies the taints using a taint analysis tool built on top of Valgrind [17, 18, 22], and applies the analysis results back to refine the process recording.

The API has two tainting operations: $taint(A)$ taints a piece of data A, $untaint(A)$ removes taint on a piece of data. Untainting balances the need for privacy with the desire to preserve an interesting, debuggable recording. It can allow us to retain more of the original recorded computation in the final redacted recording. This may be desirable for example, when computing a cryptographic one-way function on sensitive data: though the result will be tainted, we may choose to untaint it because the output preserves the privacy of the source data.

Crosscut ensures the absence of tainted data in the recording by time slicing the computation from a point identified by the programmer using the API (at some point before tainted data is first read) extending to the last use of the tainted data as determined by the taint analysis. The tainted memory existing after the time slice is overwritten with zeroes in the redacted recording. This doesn't impugn the determinism of replay because there are no further accesses to this memory. [2]

Multi-stage replay is an important property in taint redaction. It allows code *reuse*: a maintained, purpose-built process-based taint tracking implementation already exists in Valgrind and can be reused to track sensitive data taints, while process extraction and time slicing can run in the VMM. Splitting the operation up as a composition of reusable, specialized tools makes multi-stage replay more flexible and powerful, much like a series of commands connected via Unix pipes.

4.5 Snapshot pruning

Crosscut can prune state required in the initial snapshot of recordings to include only state actually used during the recording. For VM recordings, both the disk and main memory image can be pruned; for process slices, the process address space can be pruned.

Crosscut identifies disk blocks read and written by a VM recording by replaying the VM, whose initial snapshot includes a complete copy of the virtual disk. After the profile is collected, the recording's virtual disk can be pruned with a read-before-write rule: only disk blocks read during the recording before being written to are included in the virtual disk. If a disk block isn't read at all, or if it is written to before being read from, then they need not be included in the recording's snapshot of the disk—they either aren't needed, or they will be deterministically generated during replay.

Memory, for process or VM snapshots, must be handled differently. Memory usage can be tracked efficiently by protecting pages in the MMU. However, unlike disk blocks, memory pages are typically only partially written on a write. Consequently, the read-before-write rule doesn't work: because most writes to a page don't fully overwrite the page, the program can read data from the page that isn't satisfied by a recorded write. Therefore, memory is pruned slightly more conservatively: all accessed (read or written) pages are included in the initial snapshot for the VM or process. Pages not accessed can be eliminated.

[2] Crosscut actually implements a small variant of this scheme where the time slice extends not to the last use of the tainted data, but to the last branch that depends on a comparison on tainted data. Tainted memory can be overwritten with zeroes after the time slice, but replay determinism isn't affected because no branches occur on the tainted data.

Process recording	Log (MB)	Snapshot (MB)	Log/Relog time (sec.)	Replay time (sec.)
VM recording	31	4403	285	210.1
VM recording *pruned*	31	419	285	186.1
Internet Explorer	896	271	1766	56.2
MS Word	684	247	2127	63.4
PowerPoint	301	234	1325	36.7
verclsid.exe	3.7	95	340	4.1
ctfmon.exe	5.5	215	352	10.2
verclsid.exe	3.7	5	289	1.7
svchost.exe	6.3	134	346	4.2
verclsid.exe	3.7	97	324	2.6

Figure 3. Space and time savings from snapshot pruning and process slicing (desktop workload).

Process recording	Log (MB)	Snapshot (MB)	Log/Relog time (sec.)	Replay time (sec.)
VM recording	14.8	11610	214	226
VM recording *pruned*	14.8	725.7	214	226
IIS	68.8	13.6	470	19
php-cgi	477.9	463.8	4295	740
mysqld	90.1	135	654	220

Figure 4. Space and time savings from snapshot pruning and process slicing (server workload).

An important part of multi-stage replay is that it allows pruning expenses, such as MMU faulting, to move off the critical path of recording. To be fast, traditional two-stage record and replay conservatively overestimates what state is required in the initial snapshot of computation that starts recording: not knowing what state may be needed by the recording, all possible state is included. For a VM, this means all of main memory, or the whole virtual disk.

Tracking use of these resources during relogging allows us to prune the snapshot without paying the cost of tracking their use during recording. The snapshot pruning mechanism can also be applied to generated abstraction or time slices, since these subsetted computations typically use only a subset of the initial data.

5. Evaluation

In this section, we show how Crosscut uses multi-stage replay to leverage information about the end replay that is learned after the original recording. We show how multi-stage replay can shrink the recording to that needed during a specific replay period, slice out processes that are not relevant during replay (and may contain sensitive information), support replay at higher levels of abstraction, and ensure privacy.

We use a variety of workloads to evaluate Crosscut. To evaluate snapshot pruning and process slicing, we use two general-purpose workloads. The first workload represents a typical desktop environment, in which a Windows XP user browses the web using Internet Explorer and creates Microsoft Word and PowerPoint documents. The second workload represents a typical web server environment, in which a IIS web server runs the php-based MediaWiki application and responds to HTTP POST requests for a series of wiki pages. The MediaWiki application communicates with a MySQL database server to process the requests. We use specialized workloads to evaluate Perl slicing, offloading, and privacy redaction.

5.1 Snapshot pruning

The original recording provided by VMware Workstation reconstructs the entire state of a virtual machine. While this feature is quite powerful (e.g., it enables one to resume a replayed virtual machine and continue live execution), it requires a much larger snapshot than needed for many situations. For example, to diagnose a crash, developers would usually need only to replay a finite time interval, which can be done by reconstructing only the state read or written in that time interval. As described in Section 4.5, Crosscut can prune the state required in the initial snapshot to include only state actually needed during the recording.

We evaluate the effectiveness of snapshot pruning on a general-purpose desktop and server workload. The top sections of the tables in Figures 3 and 4 show how snapshot pruning can reduce the size needed to represent a recorded computation and the time needed to replay this computation. The unpruned snapshot for this recording includes the entire disk and memory of the virtual machine. By pruning the snapshot to include only the state needed by the recorded portion of computation, Crosscut is able to reduce the size of the snapshot by 90% in the desktop workload and 94% in the server workload. Replay times are usually shorter than the original recording because less computation is being replayed. However, replay of specific processes may be slower because injecting the results of a system call using the VMM may be slower than having the operating system re-execute the system call inside the VM.

5.2 Process slicing

We next evaluate Crosscut's multi-stage replay approach and compare it to a two-stage approach that logs at the process level. For each workload (desktop and server), we assume the end-user of replay (e.g., intrusion analysis) is interested in examining the execution of some of the processes running on the system, but that the end-user does not know *a priori* which processes are of interest and so must log them all.

In the prior two-stage approach, the system must log all processes during the original run. The bottom sections of each table in Figures 3 and 4 show that this approach requires storing and transporting a large amount of log and snapshot data, even after pruning (3202 MB for the desktop workload's log and snapshot; 1249 MB for the server workload's log and snapshot).

In contrast, Crosscut needs only to save the pruned VM log and snapshot for this time interval, which saves 86% of the log and snapshot space for the desktop workload and 41% of the log and snapshot space for the server workload. Crosscut saves space by not logging the interprocess communication between processes. Instead, Crosscut can regenerate this data by replaying the processes and the operating system. Crosscut also saves space by eliminating redundancy in the data stored multiple times in process snapshots.

While Crosscut saves space over two-stage process-level logging, it can also regenerate the same final recording that is saved by two-stage process-level logging. Crosscut can defer the work and storage needed to create the process-level log to after the original run, and Crosscut need only generate the recording for the processes that are of interest during post-execution analysis.

The cost for achieving this flexibility is that Crosscut must post-process the original recording through relogging, which takes 1.1-20x as long as the original execution time. However, the relogging step can be done offline so it does not slow the original execution.

Process slicing can also reduce the leaking of private data. Crosscut can create a slice that includes only the process that needs to be debugged, while leaving out other processes that may contain sensitive data. For example, Crosscut can create a slice containing all processes launched in a remote ssh session, but exclude the `sshd` process because it handles private encryption keys.

The privacy and size reductions may make it possible to replace core dumps with fully replayable process level logs. A 4 GB VM would take over 6 hours to send over a T1 line while raising substantial privacy concerns. Figures 3 and 4 show that the process logs generally save more than 70% compared to an unpruned VM recording (log plus snapshot) while simultaneously providing additional privacy benefits.

5.3 Perl debugging

Interpreted languages such as Perl are difficult to debug with comprehensive replay logs [10]. Accessing internal state can require trapping to the virtual machine monitor on every interpreted instruction, and this slows performance substantially. By raising the level of abstraction being replayed, we can use the Perl debugger without perturbing the Perl interpreter, and thereby allow a programmer to interact more naturally with the recorded execution.

For this particular example, we use a VM that runs SPECweb2005 and AWstats. We recorded a section of its execution during which SPECweb was executing, and both legitimate and malicious requests were dispatched to AWstats. We then extracted all of the Perl script execution into individual logs, and were able to analyze each in turn to determine first which specific instantiation of AWstats had been attacked, and finally the specific line that, as a result of improper input sanitization, had allowed the exploit to produce a file in the `/tmp` directory.

Crosscut's Perl relogger generated slices of all Perl processes in a single replay of the VM. We were able to replay the resulting Perl slices on our Perl interpreter and modify the execution of the Perl interpreter (e.g., by adding breakpoints) without perturbing the faithful replay of the Perl script. In addition, the Perl-level slices generated by Crosscut were much smaller than the original VM-level recording (7.5KB vs. several GB), which again highlights the need for multi-stage replay to achieve both comprehensive recording of the entire system and space-efficient recordings of the entities that are later of interest.

5.4 Chronicle debugging

This section evaluates how Crosscut can offload the work to generate heavyweight recordings from a logged machine to a secondary analysis machine.

Omniscient debugging [12, 20, 21] is a form of debugging where all execution history for a program invocation (including all writes to variables and all function calls) is recorded in a database, and debugging the program becomes purely a series of indexed database queries, such as "*when was the last write to location X before time T*", or "*when was location X executed between times T1 and T2.*" Fast access to this database via indexing can make this an attractive way to debug when compared to normal debugging or even traditional replay debugging.

Unfortunately, recording and indexing this execution history is extremely slow and generates vast amounts of data. Debugged programs are typically slowed down 100-300x [12, 20], and log data is generated at a rate of 100s of MB to GBs per second. This expense makes omniscient debugging impractical to use on normal, live execution.

Multi-stage replay and abstraction slicing enables omniscient debugging to be used on production runs by deferring the expense of recording and indexing until it is actually needed, such as when there's a crash to be debugged. Only the relatively small overhead of VM-level recording needs to be incurred on the production machine. Using abstraction slicing, Crosscut can take a VM-level recording, create a slice that includes only the process to be debugged, and retarget that process to a omniscient debugging platform where domain knowledge allows in-depth examination. Crosscut retargets sliced processes to run on the Chronicle debugger [20, 21], which has three parts: an indexer written as a Valgrind tool, a query engine, and a Eclipse UI front-end.

To measure the savings of this approach, we used a 32-bit Ubuntu 8.10 guest running the Apache 2.2.9 web server, with web requests handled by a CGI program. A specially crafted web request triggers a buffer overflow in the CGI program, causing it to malfunction and need debugging. In our experiment, VMware Workstation records the whole system during this execution, then Crosscut creates a slice of the failing CGI program and runs it through the normal Chronicle indexer in our replay-enabled Valgrind. Using the Eclipse UI front-end, we are able to walk backwards and forwards in the recorded execution and perform queries that pinpoint the injected overflow.

Crosscut saves substantial time and space during the original recording by deferring work to later, offline stages. With VM-level recording, the CGI script runs in less than one second. Chronicle indexing of the same execution takes 6 minutes 33 seconds and produces a database 405MB in size.

5.5 Redacting sensitive data

Searching for a specific data pattern is an unsound but commonly used method for detecting and preventing data leakage. Crosscut provides a transformer to scan a VM and remove all execution time that requires the use of memory containing sensitive state.

We validated this transformation with a pattern that would match the contents of password hashes in standard unix format. We applied our transformer with this pattern to a VM running a workload that used `/etc/shadow`. Our transformer successfully sliced out execution from when password data was returned from a system call, to when a scan of memory confirmed that it was no longer present. The resulting sliced log prevented the leak of sensitive information, while still including 99.986% of the original recording.

For a more sophisticated example of removing sensitive data, we applied our taint based redaction to remove all execution time that was spent handling a private key used by OpenSSL in a simple signing program. We observed our test program loading a private key on disk, creating a signature on some file data, freeing the private key, verify the signature using the public key, and noted that the taint system precisely identified the uses and copies of the private key and memory, and successfully removed the time slices containing tainted key data.

6. Related work

Deterministic replay has been used to record and replay the execution of many different entities in computing systems, including processors [1, 8, 15, 27], virtual machines [2, 5, 28], processes [23], Java virtual machines [16], and code that runs on libraries [7]. Each of these systems chooses a specific entity to record and makes the corresponding tradeoff between comprehensiveness on one hand, and efficiency, flexibility, and privacy on the other. In contrast, Crosscut seeks to provide the best of both worlds by recording the original run comprehensively, then transforming the log through multiple stages of replay to provide efficiency, flexibility, and privacy.

Researchers have used deterministic replay to reproduce and then analyze the execution of a system. This analysis has been done while the system is being logged [19, 25] or after the logging is complete [5]. We envision Crosscut being used primarily after the logging is complete, though it could also be used during logging. Aftersight uses deterministic replay to offload the work of analysis from the logged machine [4]. Crosscut uses this idea to offload the work of transforming the captured recording from the logged machine. Aftersight also introduced the idea of relogging and used it to remove dependencies on specific I/O devices. Crosscut applies relogging to transform the recording in many more ways, such as preserving privacy, reducing the size and replay time of the recording, and targeting different abstraction levels.

The issue of minimizing loss of privacy while providing helpful information to developers was examined by Castro, et al. [3]. They describe a system that preserves privacy by generating new user inputs that make a program follow a failed execution path. Crosscut handles privacy concerns at a coarser granularity by slicing out entities (e.g., processes, interpreters, operating systems) and time periods that contain sensitive data. [3] can be viewed as a sanitized form of deterministic replay that preserves only the execution path (rather than all data) and is concerned only with explicit program input (rather than all non-deterministic events, such as timing).

Crosscut uses replay transformers to bridge the semantic gap between different levels of replay. These replay transformers may leverage functionality that is already present in higher levels of software, as was done in the IntroVirt system [10], or may leverage hardware or operating system interfaces to reconstruct the higher level abstractions, as was done in VMwatcher [9].

Replay forking is similar to the checkpointing/rollback that IntroVirt uses to perform introspection [10]. The main difference is in how the extra code is invoked. IntroVirt triggers the code by alerting an external entity via breakpoints, which then modifies the registers to initiate a function call. Crosscut invokes the extra code by setting page protections that trigger a control transfer within the replayed domain. Crosscut's approach eliminates numerous context switches between the replayed domain and other processes.

The term *abstraction slicing* was inspired in part by past work on program slicing [26], which seeks to isolate the part of a program that influence the value of a variable.

7. Conclusions

Comprehensive deterministic replay systems provided by platforms such as virtual machine level record-replay provide many compelling properties such as low recording overhead and a complete view of system state. Unfortunately, current systems suffer from several major drawbacks. Recordings can be very large, and these make transportation and replay unwieldy and raise concerns about privacy. In addition, the "semantic-gap" between whole system replay and the process or language level that a programmer works at can hinder usability.

We presented a system called Crosscut that addresses these problems by allowing recorded computations to be "sliced" along time and abstraction boundaries before being replayed by end users. Using this approach we can transform replay logs to contain only the processes, applications, and portions of execution time that are of interest at the time of replay. Slicing recordings in this manner saves space, improves replay performance, and can exclude sensitive data. In addition, abstraction slicing allows us to retarget our recordings to different higher-level replay platforms, such as Perl or Valgrind, thus bridging the semantic gap that can hinder programmer understanding. This also allows execution to be augmented with additional analysis code at replay time, without disturbing the replayed components in the slice.

In our experiments, our current Crosscut prototype can reduce the size of an execution history by an order of magnitude, speed replay by several-fold, and substantially improves usability.

References

[1] BACON, D. F., AND GOLDSTEIN, S. C. Hardware-Assisted Replay of Multiprocessor Programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging* (May 1991).

[2] BRESSOUD, T. C., AND SCHNEIDER, F. B. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems 14*, 1 (February 1996), 80–107.

[3] CASTRO, M., COSTA, M., AND MARTIN, J.-P. Better Bug Reporting With Better Privacy. In *Proceedings of the 2008 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (March 2008), pp. 319–328.

[4] CHOW, J., GARFINKEL, T., AND CHEN, P. M. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of the 2008 USENIX Technical Conference* (June 2008), pp. 1–14.

[5] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M., AND CHEN, P. M. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation* (December 2002), pp. 211–224.

[6] GARFINKEL, T., AND ROSENBLUM, M. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the 2003 Network and Distributed System Security Symposium (NDSS)* (February 2003).

[7] GUO, Z., WANG, X., TANG, J., LIU, X., XU, Z., WU, M., KAASHOEK, M. F., AND ZHANG, Z. R2: An Application-Level Kernel for Record and Replay. In *Proceedings of the 2008 Symposium on Operating Systems Design and Implementation* (December 2008), pp. 193–208.

[8] HOWER, D. R., AND HILL, M. D. Rerun: Exploiting Episodes for Lightweight memory Race Recording. In *Proceedings of the 2008 International Symposium on Computer Architecture* (June 2008), pp. 265–276.

[9] JIANG, X., WANG, X., AND XU, D. Stealthy Malware Detection Through VMM-Based Out-of-the-Box Semantic View Reconstruction. In *Proceedings of the 2007 ACM Conference on Computer and Communications Security (CCS)* (November 2007), pp. 128–138.

[10] JOSHI, A., KING, S. T., DUNLAP, G. W., AND CHEN, P. M. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 2005 Symposium on Operating Systems Principles* (October 2005), pp. 91–104.

[11] KING, S. T., DUNLAP, G. W., AND CHEN, P. M. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the 2005 USENIX Technical Conference* (April 2005), pp. 1–15.

[12] LEWIS, B. Debugging backwards in time. In *Proceedings of the 2003 Workshop on Automated and Algorithmic Debugging* (September 2003).

[13] LITZKOW, M., AND SOLOMON, M. Supporting Checkpointing and Process Migration outside the Unix Kernel. In *Proceedings of the Winter 1992 USENIX Conference* (January 1992).

[14] MELLOR-CRUMMEY, J. M., AND LEBLANC, T. J. A Software Instruction Counter. In *Proceedings of the 1989 International Conference on Architectural Support for Programming Languages and Operating Systems* (April 1989), pp. 78–86.

[15] MONTESINOS, P., CEZE, L., AND TORRELLAS, J. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently . In *Proceedings of the 2008 International Symposium on Computer Architecture* (June 2008), pp. 289–300.

[16] NAPPER, J., ALVISI, L., AND VIN, H. A Fault-Tolerant Java Virtual Machine. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN)* (June 2003).

[17] NETHERCOTE, N., AND SEWARD, J. How to shadow every byte of memory used by a program. In *Proceedings of the 2007 Conference on Virtual Execution Environments* (June 2007).

[18] NETHERCOTE, N., AND SEWARD, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 Programming Language Design and Implementation Conference* (June 2007).

[19] NIGHTINGALE, E. B., PEEK, D., CHEN, P. M., AND FLINN, J. Parallelizing security checks on commodity hardware. In *Proceedings of the 2008 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (March 2008), pp. 308–318.

[20] O'CALLAHAN, R. Efficient collection and storage of indexed program traces. `http://www.ocallahan.org/Amber.pdf`, December 2006.

[21] O'CALLAHAN, R. Announcing chronomancer. `http://weblogs.mozillazine.org/roc/archives/2007/08/announcing_chro.htm%l`, August 2007.

[22] SEWARD, J., AND NETHERCOTE, N. Using Valgrind to detect undefined value errors with bit-precision. In *Proceedings of the 2005 USENIX Technical Conference* (April 2005).

[23] SRINIVASAN, S., KANDULA, S., ANDREWS, C., AND ZHOU, Y. Flashback: A light-weight rollback and deterministic replay extension for software debugging. In *Proceedings of the 2004 USENIX Technical Conference* (June 2004).

[24] TUCEK, J., LU, S., HUANG, C., XANTHOS, S., AND ZHOU, Y. Triage: Diagnosing Production Run Failures at the User's Site. In *Proceedings of the 2007 Symposium on Operating Systems Principles* (October 2007), pp. 131–144.

[25] WALLACE, S., AND HAZELWOOD, K. SuperPin: Parallelizing Dynamic Instrumentation for Real-Time Performance. In *Proceedings of the 2007 International Symposium on Code Generation and Optimization (CGO)* (March 2007), pp. 209–217.

[26] WEISER, M. Programmers use slices when debugging. *Communications of the ACM 25*, 7 (July 1982), 446–452.

[27] XU, M., BODIK, R., AND HILL, M. D. A Flight Data Recorder for Enabling Full-system Multiprocessor Deterministic Replay. In *Proceedings of the 2003 International Symposium on Computer Architecture* (June 2003).

[28] XU, M., MALYUGIN, V., SHELDON, J., VENKITACHALAM, G., AND WEISSMAN, B. ReTrace: Collecting Execution Trace with Virtual Machine Deterministic Replay. In *Proceedings of the 2007 Workshop on Modeling, Benchmarking and Simulation (MoBS)* (June 2007).