Multi-stage replay with Crosscut

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

Peter M. Chen
University of Michigan
Analyzing with records

• Detailed recordings of program execution
  – Program output, logs
  – syslogd
  – Ad-hoc records (“printf” debugging)
  – Backtraces
  – Core dumps

• Useful for analyzing programs
  – Correctness, performance, security
Debugging programs with records

- So useful, now commonplace to send records back to the developers:
Debugging programs with records

- Despite extensive collection of records, programs still hard to debug
  - Program output, logs
  - syslogd
  - Ad-hoc records ("printf" debugging)
  - Backtraces
  - Core dumps
- Don’t know everything about a program
- Miss behavior: don’t run all the time, or over all values---must decide **when** or **what** to record (balancing act)
Instead: record everything, all the time

• Easy to do with a VMM
• VMM: thin layer between software stack and HW
• All inputs into system must pass through the VMM
  – All possible decisions governed only by these inputs
Alternative: VM record/replay

• Easy to do with a VMM
• VMM: thin layer between software stack and HW
• All inputs into system must pass through the VMM
  – All possible decisions governed only by these inputs

• If we save inputs
  – and their timing
  – have all non-determinism we need
Alternative: VM record/replay

• Easy to do with a VMM
• VMM: thin layer between software stack and HW
• All inputs into system must pass through the VMM
  – All possible decisions governed only by these inputs

• If we save inputs
  – and their timing
  – have all non-determinism we need
• To replay:
  – Start from clean state
Alternative: VM record/replay

Some nice properties:
Complete *coverage*: all data, all the time, app/OS agnostic
Alternative: VM record/replay

Some nice properties:
Complete **coverage**: all data, all the time, app/OS agnostic

Low, paradoxical performance perturbation:
5% overhead
SPEC CPU2005
Alternative: VM record/replay

Some nice properties:
Complete coverage: all data, all the time, app/OS agnostic

Low, paradoxical performance perturbation:
5% overhead SPEC CPU2005

Low bandwidth requirements: network input + epsilon (KB/s)
Alternative: VM record/replay

But have to replay to reclaim state:
- Need a replayer.
  - Input not understandable
- Need to feed all inputs to replayer to get any useful output.
  - Won’t work with subset.

Again:
- You need a VMM
  - To build tools, must modify VMM
    - Hard to do: not user-level
    - Can be slow
- You need a VM
  - Clunky: to debug one program, you need all programs, data in VM.
Goal: replay with discretion

• VM recordings are great.
  – Provide great coverage.

• But we want to replay with discretion:
  – Don’t require/expose all inputs (no VM).
  – Don’t require same abstractions (no VMM).
  – Don’t sacrifice coverage for discretion.

• How are we going to do this?
VM replay is a state machine

- Unhappy one size fits all
- Convert inputs to a new machine
  - Represents some computation in original
Generate abstractions

- Know we can generate new machine for any abstract state, computation in the log:
  - Program output, logs
  - syslogd
  - Ad-hoc records ("printf" debugging)
  - Backtraces
  - Core dumps

- **VM recordings subsume all of the above.**
  - Any data we need can be generated.
Fixing replay problems with slices

In other words:
• Instead of VM log:
  – Generate a unit of data, slice
  – Slice is abstracted input for new machine
Fixing replay problems with slices

In other words:
• Instead of VM log:
  – Generate a unit of data, slice
  – Slice is abstracted input for new machine
Fixing replay problems with slices

A slice will:
• Retain useful information needed for replay debugging or analysis
Fixing replay problems with slices

A slice will:
• Retain useful information needed for replay debugging or analysis

And it will:
• Not require a copy of the VM
• Not require a VMM to replay it
Fixing replay problems with slices

A slice will:
• Retain useful information needed for replay debugging or analysis

And it will:
• Not require a copy of the VM
• Not require a VMM to replay it

A slice is *any* higher-level recording destined for a higher-level state machine.
What are some useful state machines?

• Motivator:
  – Replay debugging for Dr. Watson-type crash dumps
This machine is weird

• Motivator:
  – Replay for Dr. Watson-type crash dumps
  – Requires whole VM
    • Fine within an organization
    • Silly otherwise
A better machine

- Generate inputs for new abstract machine
  - Replays just a process
- Given inputs, cannot generate state for other processes
  - Better than original machine
How to represent process state

- Retain benefits of original system:
  - Record all input to a process
    - Including timing, races, shared memory accesses (*)
      - (*) at least for VMs that are virtual uniprocessors.
  - Agnostic to application, OS
Choose a representation independent of OS

- Represented with HW state.
  - Address space, registers, memory map, no OS state.
- Given representation, runs deterministic
  - Until needs input: syscall, signals, CPU counters, etc.
  - Input’s only effect: modify represented state
  - Continually supply this state.
Figure out where state comes from

Normal system calls, traps, interrupts

- Only comes from 3 places
- Modified VMM
  - Hook sites of introduction
  - Control flow into/out of process

Application

Operating system

System calls, traps, interrupts when slicing a process

Application

Operating system

VMM

Out flow  In flow
Collect input without OS help

System calls, traps, interrupts when slicing a process

- OS injects state (directly on indirectly)
  - Do it with MMU: agnostic to OS.
  - Notice modifications to sliced process’s address space when it’s not running
How to detect writes

- System calls, traps, interrupts when slicing a process

- Application

- Operating system

- VMM

  • OS injects state (directly on indirectly)
    - Do it with MMU: agnostic to OS.
    - Notice modifications to sliced process’s address space when it’s not running
Slices are standalone recordings:
- Fully capable
- Discrete
- Not specific to OS/VMM
  - Why not replay at user-level?
High level logs are useful

- High level log: any user-level tool can be a replayer.
- User-level a big thing:
  - Lots of user-level tools for analyzing “live” programs.
  - Crosscut supports running slices in valgrind, Omniscient debugger, Chronicle. Standard tools, gdb and visual studio.

High level log: Even Windows proc on valgrind.
Beyond processes

• Processes not only abstraction
  – Sometimes not useful at all.
Beyond processes

- Processes not only abstraction
  - Sometimes not useful at all.

- Debugging the interpreter, not the script (cannot change interpreter behavior)
We can solve this with yet another better machine

- Goal: create abstraction of script execution.
  - Avoid details of underlying runtime.
Recording a script generically is not possible

• Recording a script generically is not possible.
  – Requires intimate knowledge of the interpreter.

• Crosscut provides an API:
  – Given to guest software to create their own inputs to their own machines.
  – Guest doesn’t really need an API just to create slices
    • What’s wrong with write()?
  – Purpose:
    • Avoid overhead in the guest for creating the slice. How?
    • Do regular VM recording
      – Defer slicing to if/when it’s needed.
      – Because ex-post-facto, original recording fast
      – Can be expensive---off the critical path.
What does the API do?

- Guest has code it wants to run.
- API provides notification to VMM of:
  - work (guest code) we’d like to do
  - ... at a location we want to do it
  - On replay, VMM executes the guest code in sandbox.
- Notification mechanism must be fast:
  - It is running during original recording.
  - Standard signalling mechanism: instruction to trigger faults
    - Triggering faults in original recording too slow
      - Regular hypercall mechanisms out: port I/O, #UD
    - Silent/fast during recording: memory references, MMU
How to use API to slice a script

Normal script transitions

- What inputs do we want to save?
  - How to represent them?
- Modified Perl
  - Make deterministic recording of a script
What inputs do we want to save?

- Simplistic example: system calls.
- Should be sufficient
  - Calls into Perl runtime will result in calls to the system. If not, they are deterministic.
  - Should be able to replay calls to runtime from system call inputs.
How to use API to save syscalls

On occasion of slicing a script: intercepting inputs

- Modified Perl runtime:
  - Record system call inputs (deferred in sandbox).
- Isn’t this a process replay log?
- The difference: not all system calls belong to script.
  - Interpreter initiates some on its own behalf.
  - Don’t save those: want interpreter’s behavior capable of change
Want the interpreter to run free

On occasion of slicing a script: intercepting inputs

• When replaying, interpreter runs freely
  – It can change its behavior, because it’s live.
  – When script requests input, fed from log
    • Behaves deterministically
What can you do with Perl log?

- Created a new machine: Perl interpreter + replay mechanism
- Internal Perl facilities can be used to debug the script: Perl debugger can set breakpoints, inspect variables.
- A better machine: let Perl interpreter do debugging, more natural than stapling Perl introspection into a VMM.
Goal: record program state

• VM recordings are great.
  – Provide great coverage.

• But we want to replay with discretion:
  – Don’t require VM.
  – Don’t require VMM.
  – Don’t sacrifice coverage for discretion.

• Obtain discretion with slices.
  – Abstracts replay requirements.
  – Doesn’t sacrifice coverage.
fin.