RESim User's Guide

Reverse Engineering heterogeneous networks of computers through external dynamic analysis

April 29, 2019

Contents

1 Introduction

Imagine you would like to analyze the processes running on computers within a system, including the programs they execute and the data they consume and exchange. And assume youd like to perform this analysis dynamically, but without ever running your own software on those systems and without ever having a shell on the systems.

RESim is a dynamic system analysis tool that provides detailed insight into processes, programs and data flow within networked computers. RESim simulates networks of computers through use of the Simics[1] platform's high fidelity models of processors, peripheral devices (e.g., network interface cards), and disks. The networked simulated computers load and run targeted software copied from disk images extracted from the physical systems being modeled.

Broadly, RESim aids reverse engineering of networks of Linux-based systems by inventorying processes in terms of the programs they execute and the data they consume. Data sources include files, device interfaces and inter-process communication mechanisms. Process execution and data consumption is documented through dynamic analysis of a running simulated system without installation or injection of software into the simulated system, and without detailed knowledge of kernel hosting the processes.

RESim also provides interactive visibility into individual executing programs through use of the IDA Pro disassembler/debugger to control the running simulation. The disassembler/debugger allows setting breakpoints to pause the simulation at selected events in either future time, or past time. For example, RESim can direct the simulation state to reverse until the most recent modification of a selected memory address. RESim lets the analyst switch between different threads within a process. Reloadable checkpoints may be generated at any point during system execution. A RESim simulation can be paused for inspection, e.g., when a specified process is scheduled for execution, and subsequently continued, potentially with altered memory or register state. The analyst can explicity modify memory or register content, and can also dynamically augment memory based on system events, e.g., change a password file entry when read by the `su` program (see F.4.

Analysis is performed entirely through observation of the simulated target system's memory and processor state, without need for shells, software injection, or kernel symbol tables. The analysis is said to be *external* because the analysis observation functions have no effect on the state of the simulated system.

2 Analysis artifacts

RESim generates system traces of all processes on a computer, starting with system boot, or at a selected checkpoint. Trace reports include two components:

1. A serialized record of system calls, identifying the calling process and selected parameters, e.g., names of files and sockets and IP addresses.

2. A process family history for each process and thread that has executed, identifying:

   (a) Providence, i.e., which process created the process (or thread), and what programs were loaded via the `execve` system call.

   (b) Files and pipes that had been opened (including file descriptors inherited from the parent), and those that are currently open.

   (c) Linux socket functions, e.g. , connect, accept, bind, etc. Socket connect attempts to external components are highlighted, as are externally visible socket accepts.

   (d) Mapped memory shared between processes

The system trace is intended to help an analyst identify programs that consume externally shaped data. Such programs can then be analyzed in depth with the dynamic disassembler. [TBD expand to support decompilers where available].

3 Dynamic analysis of programs executing in their environment

RESim couples an IDA Pro2 disassembler debugger client with the Simics simulation to present a dynamic view into a running process. The analyst sets breakpoints and navigates through function calls in both the forward and reverse execution directions. This facilitates tracking the sources of data. For example, if a program is found to be consuming data at some location of interest, reverse execution might identify a system call that brings the data into the processs address space.

[1] Simics is a full system simulator sold by Intel/Wind River, which holds all relevant trademarks.

A key property that distinguishes RESim from other RE strategies is that analysis occurs on processes as they execute in their native environment, and as they interact with other processes and devices within the system. Consider an example process that communicates with a remote computer via a network while also interacting with a local process via a pipe. When the analyst pauses RESim for inspection, the entire system pauses. The simulation can then be resumed (or single-stepped) from the precise state at which it was paused, without having to account for timeouts and other temporal-based discontinuities between the process of interest and its environment.

4 Limitations

RESim analyzes Linux-based systems for which copies of bootable media or root file systems can be obtained. Analysis does not depend on a system map of the kernel, i.e., it works with stripped kernel images. The current version of RESim supports 32-bit and 64-bit X86 and 32-bit ARM. It can also be extended to support alternate architectures, e.g., 64-bit ARM, supported by Simics processor models [2].

5 Work Flow

Simulated systems are defined within RESim configuration files that parameterize pre-defined Simics scripts to identify processors and interface devices, e.g., network cards, disks and system consoles. RESim currently includes a generic X86 platform and a 32-bit ARM platform. Other platforms can be modeled via Simics, the detailes of which are beyond the scope of this manual. Once a system is modeled and referenced by a RESim configuration file, a RESim script is run, naming the configuration file.

RESim analysis requires about twenty parameters that characterize the booted kernel instance, e.g., offsets within task records and addresses of selected kernel symbols. The `getKernelParams` utility automatically analyzes a running kernel and extracts the parameters required by RESim. This utility allows RESim to analyze disparate Linux kernels without a priori knowledge of their versions or configurations [3].

Once the kernel parameters have been extracted, the simulation boots under control of RESim, which then presents the user with a command line interface console. This console manages the simulation via a combination of RESim and Simics commands, including commands to:

- Start or stop (pause) the simulation
- Run until a specified process is scheduled
- Run until a specified program is loaded, i.e., via execve
- Generate a system trace
- Inspect memory and component states
- Set and run to breakpoints
- Enable reverse execution, i.e., allow reversing from that point forward
- Target the RESim disassembler/debugger for a specified process

When RESim is targeted for a given process it runs until that process is scheduled, after which the user starts IDA Pro with a suite of custom plugins that interact with the simulation. If a binary image of the target program is available, standard IDA Pro analysis functions are performed. If no program image is available, IDA Pro will still present disassembly information for the program as it exists in simulated system memory.

RESim extends IDA Pro debugger functions and the analyst accesses these functions via menus and hot keys. The disassembler/debug client can be used to:

- Single-step through the program in either the forward or reverse direction
- Set and run to breakpoints in either direction
- Run to the next (or previous) system call of a specified type, e.g., open.
- Run to system calls with qualifying parameters, e.g., run until a socket connect address matches a given regular expression.

[2] A summary of Simics device models is at: `https://www.windriver.com/products/simics/simics-supported-targets.html`

[3] Not to be confused with the similar function included with Simics Analyzer product. RESim uses an alternate strategy for OS-Awareness.

- Reverse-trace the source of data at a memory address or register.
- Modify a register or memory content
- Switch threads of a multithreaded application
- Set and jump to bookmarks

6 Development and Availability

In addition to running on a local Simics installation, RESim is intended to be offered as a network service to users running local copies of IDA Pro and an SSH session with a RESim console. The tool is derived from the Cyber Grand Challenge Monitor (CGC), developed by the Naval Postgraduate School in support of the DARPA CGC competition. RESim is implemented in Python, primarily using Simics breakpoints and callbacks, and does not rely on Simics OS Awareness or Eclipse-based interfaces. IDA Pro extensions are implemented using IDAPython. All of the RESim code is available on github at `https://github.com/mfthomps/RESim`

7 RESim commands

- `traceProcesses` Begin tracing the following system calls as they occur: vfork; clone; execve; open; pipe; pipe2; close; dup; dup2; socketcall; exit; group_exit Tracing continues until the stopTrace command is issued.
- `runTo` Continue execution until the named program is either loaded via execve or scheduled. Intended for use prior to tracing processes, e.g., to get to some known point before incurring overhead associated with tracing. This function will track processes PIDs and names along with network configuration information, and will save that data if a writeConfig function is used.
- `writeConfig` Uses the Simics write-configuration command to save the simulation state for later loading with read-configuration. This wrapper also saves process naming information and network configuration commands for reference subsequent to use of the read-configuration function.
- `traceAll` Begin tracing all system calls. If a program was selected using debugProc as described below, limit the reporting to that process and its threads.
- `showProcTrace` Generate a process family summary of all processes that executed since the traceProcess (or traceAll) command.
- `showNets` Display network configuration commands collected from process tracing and the use of toProc.
- `showBinders` Display programs that use bind and accept socket calls intended for use during process tracing to identify processes that listen on externally accessible sockets.
- `traceFile(logname)` Copy all writes that occur to the given filename. Intended for use with log files. Output is in /tmp/[basename(logname)]
- `traceFD(FD)` Copy all writes that occur to a given FD, e.g., stdout. Output is in /tmp/output-fd-[FD].log
- `debugProc(process name)` Initiate the debugger server for the given process name. If a process matching the given name is executing, system state advances until the process is scheduled. If no matching process is currently executing, execution proceeds until an execve for a matching process. If a copy of the named program is found on the RESim host, (i.e., to read its ELF header), then execution continues until the text segment is reached. RESim tracks the process as it maps shared objects into memory (see Appendix C). The resulting map of shared object library addresses is then available to the user to facilitate switching between IDA Pro analysis and debugging of shared libraries and the originally loaded program.

  Subsequent to the debugProc function completion, IDA Pro can be attached to the simulator for a SIMDis session. Most of the commands listed below have analogs available from within IDA Pro.

  If execution transfers to a shared object of interest, the associated library file can be found via the getSOFile command described below. Loading that file into IDA Pro (with the SIMDis plug-ins) will cause IDA to rebase to the address at which the shared object was mapped in the process.
- `runToSyscall(call number)` Continue execution until the specified system call is invoked. If a value of minus 1 is given, then any system call will stop execution. If the debugger is active, then execution only halts when the debugged process makes the named call.

- `runToConnect/runToAccept(search pattern)` Continue execution until a socket connection to an address matching the given search pattern.
- `runToIO(fd)` Continue execution until a read or write to the given file descriptor.
- `clone(nth)` Continue execution until the nth clone system call in the current process occurs, and then halt execution within the child.
- `runToText()` Continue execution until the text segment of the currently debugged process is reached. This, and revToText, are useful after execution transfers to libraries, or Linux linkage functions, e.g., references to the GOT.
- `revToText()` Reverse execute the current process until the text segment is reached.
- `showSOMap(pid)` Display the map of shared object library files to their load addresses for the given pid.
- `getSOFile(pid, addr)` Display the file name and load address of the shared object at the given address.
- `revInto` Reverse execution to the previous instruction in user space within the debugged process.
- `revOver` Reverse execution to the previous instruction in the debugged process without entering functions, e.g., any function that may have returned to the current EIP.
- `uncall` Reverse execution until the call instruction that entered the current function.
- `revToWrite(address)` Reverse execution until a write operation to the given address within the debugged process.
- `revToModReg(reg)` Reverse execution until the given register is modified.
- `revTaintReg(reg)` Back trace the source of the content the given register until either a system call, or a non-trivial computation (for evolving definitions of non-trivial.
- `revTaintAddr(addr)` – Back trace the source of the content the given address until either a system call, or a non-trivial computation
- `runToUser()` Continue execution until user space of the current process is reached.
- `runToDiddle()` Run until a specified write system call is encountered, and modify system memory such that the result of the write is augmented by a named diddle directive file.
- `runToDiddleRead()` Run until a specified read system call is encountered, and modify system memory such that the result of the read is augmented by a named diddle directive file.
- `reverseToUser()` Reverse execution until user space of the current process is reached.
- `setDebugBookmark(mark)` set a bookmark with the given name.
- `goToDebugBookmark` jump to the given bookmark, restoring execution state to that which existed when the bookmark was set.
- `watchData` run forward until a specified memory area is read. Intended for use in finding references to data consumed via a read system call. Data watch parameters are automatically set on a read during a debug session, allowing the analyst to simply invoke the watchData function to find references to the buffer.
- `getStackTrace` shows the call stack as seen by the monitor. The Ida client uses this to maintain its view of the callstack.
- `catchCorruption` Watch for events symptomatic of memory corruption errors, e.g., SEGV or SIGILL exceptions resulting from buffer overflows. This is automatically enabled during debug sessions. Refer to Appendix B for information about what we mean by SEGV, and how we catch it.

8 Defining a target system

This section assumes some familiarity with Simics. RESim is invoked from a Simics workspace that contains a RESim configuration file. This configuration file identifies disk images used in the simulation and defines network MAC addresses. The file uses `ini` format and has at least two sections: and `ENV` section and one section per computer that will be part of the simulation. An example RESim configuration file is at:

`$RESIM/simics/workspace/ubuntu.ini`

8.1 ENV section

The following environment variables are defined in the `ENV` section of the configuration file:

- `RUN_FROM_SNAP` The name of a snapshot created via the `@cgc.writeConfig` command.
- `OS_TYPE` Either LINUX or LINUX64
- `RESIM_TARGET` Name of the host that is to be the target of RESim analysis. Currently only one host can be analyzed during a given RESim session.
- `CREATE_RESIM_PARAMS` If set to `YES`, the getKernelParams utility will be run instead of the RESim monitor. This will generate the Linux kernel parameters needed by RESim. Use the `@gkp.go()` command from the Simics command prompt to generate the file.
- `RESIM_PARAM` Name of a parameter file created by getKernParams utility.
- `RESIM_UNISTD` Path to a Linux `unistd*.h` file that will be parsed to map system call numbers to system calls.
- `RESIM_ROOT_PREFIX` Path to the root of a file system containing copies of target executables. This is used by RESim read elf headers of target software and to locate analysis files generated by IDA Pro.

8.2 Target sections

Each computer within the simulation has its own section containing the following:

- `SIMICS_SCRIPT` Path to the Simics script file that defines the target system. This path is relative to the `target` directory of either the workspace, or the RESim repo under `simics/simicsScripts/targets`. For example,

  ```
  SIMICS_SCRIPT=x86-x58-ich10/genx86.simics
  ```

  would use the generic X86 platform distributed with RESim.
- `host_name` Name to assign to this computer.
- `use_disk2` Whether a second disk is to be attached to computer.
- `use_disk3` Whether a third disk is to be attached to computer.
- `disk_image` Path to the boot image for the computer.
- `disk_size` Size of disk_image
- `disk2_image` Path to the 2nd disk
- `disk2_size` Size of 2nd disk_image
- `disk3_image` Path to the 3nd disk
- `disk3_size` Size of 3nd disk_image
- `mac_address` Enclose in double quotes
- `mac_address_2` Enclose in double quotes
- `mac_address_3` Enclose in double quotes

8.3 Network definitions

Three network switches are created. By default, computer ethernet interface is connected to a switch, in the order of definition. TBD: add `NETWORK` section to the configuration file?

8.4 Driver component

Each network can have an optional driver component – designated by assing the string `driver` to the corresponding section header. This component will be created first, and other components will not be created until the driver has caused a file named `driver-ready.flag` is created within the workspace directory. Use the Simics Agent to create that file from the driver computer. This requires you copy the simics agent onto the target and get it to run upon boot. It is intended that the agent will load scripts to generate traffic for the target computers.

9 Running the simulation

RESim sessions are started from the Simics workspace using the $RESIM/simics/monitorCore/launchRESim.py program, where `$RESIM` is a path to the RESim repo. That program requires some environment variables, which are typically set in a bash script, an example of which is in

`$RESIM/simics/workspace/monitor.sh`

Modify that script to name the path to your RESim repo.

9.1 Getting started

Steps to define and run a RESim simulation are listed below. It is assumed you are familiar with basic Simics concepts and have a computer upon which Simics is installed with a x86-x58-ich10 platform.

1. Create a Simics workspace.
2. Copy files from `$RESIM/simics/workspace` into the new workspace.
3. Clone the RESim repo
4. Modify the monitor.sh script to reflect your Simics installation and the path to your RESim repo.
5. Modify the mytarget.ini as follows:
   - Set the `disk_image` entry to name paths to your target disk image.
   - Obtain the `unistd_32.h` or equivalent, for your target's kernel – this is used match system call numbers to calls. Name the file in the `RESIM_UNISTD` parameter.
   - Copy the target systems root file system, or a subset of the file system containing binaries of interest to the local computer and name that path in the `RESIM_ROOT_PREFIX` parameter. These images are used when analyzing specified programs, and are given to IDA Pro for analysis.
   - Set the `CREATE_RESIM_PARAMS` parameter to YES so that the first run will create the kernel parameter file needed by RESim.
6. Launch RESim using `./monitor.sh mytarget`. That will start Simics and give you the Simics command prompt.
7. Continue the simulation until the kernel appears to have booted, then stop.
8. Use the `@gkp.go()` command to generate the parameter file. This may take a while, and may require nominial interaction with the target system via its console, e.g., to schedule a new process.
9. After the parameters are created, quit Simics and remove the `CREATE_RESIM_PARAMS` parameter.
10. Restart the monitor.sh. RESim will begin to boot the target and pause once it has confirmed the current task record. You may now use RESim commands listed in 7.

9.2 IDA Pro

Once you have identified a program to be analyzed, e.g., by reviewing a system trace, open the program in IDA Pro at the location relative to the `RESIM_ROOT_PREFIX` path named in the RESim configuration file. Then, from IDA, run this script:

`$RESIM/simics/ida/dumpFuns.py`

This will create a data file used by RESim when generating stack traces.

From the Simics command line (after starting RESim), run the `@cgc.debugProc<program>` command, naming the program of interest. RESim will continue the simulation until the program is execed and execution is transfered to the text segment, at which point it will pause. Using `debugProc` rather than `debugPid` allows RESim to collect shared object information for the target process. You may now attach the IDA gdb debugger to the process (be sure to set the IDA debug port to that displayed at the Simics command prompt.) After the debugger has attached, run this IDA plugin:

```
$RESIM/simics/ida/rev.py
```

You can now run the commands found in the debugger help menu. Note those commands generally invoke RESim commands listed in 7.

A Analysis on a custom stripped kernel

Use of external analysis, (i.e., observation of system memory during system execution, to track application processes), requires some knowledge of kernel data structures, e.g., the location of the current task pointer within global data. While this information can be derived from kernel symbol tables, some systems, e.g., purpose-built appliances, include only stripped kernels compiled with unknown configuration settings.

Within 32-bit Linux, the address of the current task record can be found either within a task register (while in user mode), or relative to the base of the stack while in kernel mode. Heuristics can then be used to locate the offsets of critical fields within the record, e.g., the PID and comm (first 16 characters of the program name). While the current task record provides information about what is currently running it cannot be efficiently used to determine when the current task has changed. For that, the RESim tool prefers to know the address of the pointer to the current task record, i.e., the address of the kernel data structure that is updated whenever a task switch occurs.

Once we have the address of the current task record, a brute force search is performed starting at 0xc1000000, looking for that same value in memory. This search resulted in two such addresses being found, and use of breakpoints indicate the one at the higher memory location is updated first on a task switch.

On 64-bit Linux kernels, the current task pointer is maintained in GS segment at some processor-specific offset. This offset is not easily determined even from source code (see the arch/x86/include/percpu.h use of this_cpu_off). A crude but effective strategy for determining the offset into GS is to catch a kernel entry, and then step instructions looking for the gs: pattern in the disassembly. The first occurrence of mov rax,qword ptr gs:[seems to be the desired offset. It is expected that this will vary by cpu the getKernelParams utility needs to be updated for multi-processor (or multicore) systems.

Once the address containing the pointer to the current task record address is located, the `getKernelParam` utility uses hueristics and brute force to find the remaining parameters.

B Detecting SEGV on a stripped Linux Kernel

This note summarizes a strategy for catching SEGV exceptions using Simics while monitoring applications on a stripped kernel, i.e., where no reliable symbol table exists and /proc/kallsyms has not been read. In other words, this strategy does not rely on detecting execution of selected kernel code, e.g., signal handling.

Simics can be trivially programmed to catch and report processor exceptions, e.g., SIGILL. However, the hardware SEGV exception does not typically occur in the Linux execution environment. Rather, a page fault initiates a sequence in which the kernel concludes that the task does not have the referenced memory address allocated, and thus terminates the task with a SEGV exception.

When a page fault results from a reference to properly allocated memory in Linux, there is no guarantee that the referenced address has a page table entry. In other words, alloc does not immediately update page table structures –it is lazy. Thus, lack of a page table entry at the time of the fault is no indication of a SEGV exception. Our strategy must therefore account for modifications to the page table.

When a page fault occurs, we check the page table for an associated entry. If there is not an entry, then we set a breakpoint (and associated callback) on the page table entry, or the page directory entry if that is missing. We also locate the task record whose next field points to the faulting process, and set a breakpoint on the address of the next field. If the fault causes a page table update, it is assumed the memory reference is valid. On the other hand, if a modification is made to the next field before a page table update occurs, we assume the modification is part of task record cleanup due to a SEGV error.

C External tracking of shared object libraries

During dynamic analysis of a program, the program may call into a shared object library, and the user may wish to analyze the called library. This note summarizes how RESim provides the user with information about shared object libraries, e.g., so that the target library can be opened in IDA Pro to continue dynamic analysis. This strategy does not require a shell on the target system, nor does it require knowledge that depends on a system map, e.g., synthesizing access to /proc/¡pid¿/map

When the program of interest is loaded via an execve system call, breakpoints are set to catch the open system call. The resulting callbacks look for opening of shared library files, i.e., *.so.*. When shared objects are opened, breakpoints are then set to catch the next use of mmap by the process. We assume the resulting allocated address is where the shared object will be loaded. Empirical evidence indicates this simple brute force strategy works. These breakpoints and callbacks persist until the process execution reaches the text segment of the program.

RESim maintains maps of addresses of shared library files. When a shared object is called, the IDA Pro client retrieves the shared library name from the RESim server and displays it for the user. When the user

opens that file in IDA Pro, and attaches the debugger, the IDA Pro plug-in retrieves the address of the library and causes IDA to rebase using that offset.

D Analysis of programs with crude timing loops

Consider a program that reads data from a network interface by first setting the socket to non-blocking mode and then looping on a read system call until 30 seconds have expired. The program spins instead of sleeping. It calls "read" and "gettimeofday" hundreds of thousands of times.

Creating a process trace on such a program could take hours (or days) because the simulation breaks and then continues on each system call. This note describes how RESim identifies this condition as it occurs, and semi-automated steps it takes to disable system call tracing until the offending loop is exited.

While tracing system calls of a process, invocations of "readtimeofday" are tracked and compared to a frequency threshold. When it appears that the program is spinning on a clock, the user is prompted with an option to attempt to exit the loop. If the user so chooses, RESim will step through a single circuit of the timing loop, recording instructions at the outermost level of scope. It then searches the recorded instructions to identify all conditional jump instructions, and their destinations. Each destination is inspected to determine if it was encountered within the loop. If not, the destination and the comparison operator that controlled the jump is recorded. Breakpoints are set on each such destination address. We then disable all other breakpoints, e.g., those involved in tracing and context management, and run until we reach a breakpoint.

RESim includes an optional function to ensure that the number of breakpoints does not exceed 4 (the quantity of hardware breakpoints supported by x86). If more than 4 breakpoints are found the analyst can guide the removal of breakpoints. RESim will automatically execute the loop a large number of times in order to identify comparisons that may be converging. And the user is informed of those to aid the reduction of the quantity of breakpoints. (Note that in the context of this issue, there are less than or equal to 4 breakpoints and more than 4. There is probably a lot more than 4 as well, but we've not yet quantified its effects.)

E Breakpoints can be complicated: Real and virtual addresses

Use of a full system simulator enables *external* dynamic analysis of the system. The analysis is said to be external because the analysis mechanisms implementation, e.g., the setting of breakpoints, does not share processor state with the target. A distinguishing property of external dynamic analysis is that the very act of observation has no effect on the target system. This lack of shared effects improves the real-world fidelity of the observed system, but it can also complicate the analysis, particularly when referencing virtual addresses.

This property of external analysis is illustrated by tracing an open system call. Assume the simulator is directed to break on entry to kernel space. At that point, we can observe the value of the EAX register and determine if it is an open call. We can then observe and record the parameters given to the open system call by the application. However, the name of the file to open is passed indirectly, i.e., the parameters contain an address of a string. How might we record the file name rather than just its address?

Requesting the simulator to read the value at the given virtual address of the file name will not always yield the file name because the physical memory referenced by the virtual address may not yet have been paged into RAM by the target operating system. If the analysis were not external, then the mere reference to the virtual address could result in the operating system mapping the page containing the filename. An external analysis has no such side effects.

The simulator includes different APIs for reading virtual memory addresses and reading physical memory addresses. The former mimics processor logic for resolving virtual addresses to physical addresses based on page table structures. Attempts to read virtual addresses that do not resolve to physical addresses result in exceptions reported by the simulator – they do not generate a page fault.

Waiting to read from the file name's virtual address until after the kernel has completed the system call, i.e., until the kernel is about to return to user space, would ensure the virtual address containing the string will have been paged in. However, that strategy is susceptible to a race condition in which the file name is changed after the kernel has read it but before the trace function records it. This may occur if the file name is stored in writable memory shared between multiple threads, and could result in a trace function failing to record the correct file name used in an open system call.

Since we know the kernel will have to read the file name in order to perform the open function, we can set a breakpoint on the virtual address of the file name, and then let the simulation continue. When the kernel does reference the address, the simulation will break. In some implementations, the memory will still not have been paged in, (e.g., the kernel's own reference to the address generates a page fault), but leaving the breakpoint in place and continuing the simulation will eventually allow us to read the file name as the kernel is itself reading it. Except, that is true for only for 32-bit kernels. 64-bit kernels only reference physical addresses when reading

filenames from pages that had not been present at the time of the system call. In other words, in 64-bit Linux, breakpoints may never be hit when set on the virtual address of a file name referenced in an open call.

Even though the kernel is able to read the file name without ever referencing its virtual address, the kernel does need to bring the desired page into physical memory so that it may read the file name. In doing so, the kernel updates the page tables such that references to the virtual memory address will lead to the file name in physical memory even though such a reference may never happen. RESim takes advantage of the kernels page table maintenance by setting breakpoints on the paging structures referenced by the virtual address. In some cases, the target page table may not be present at the time of the system call, so we must first break on an update to a page directory entry, and then later break on an update to the page table entry, finally yielding address of the page in physical memory.

Note this property of the 64-bit Linux implementaion has implications beyond tracing of system calls. A reverse engineer may wish to dynamically observe the opening of a particular file name observed within an executable image. Setting a "read" breakpoint on the virtual address of the observed file name would fail to catch the open function on 64-bit Linux, while it would have caught the open on 32-bit Linux.

F Divergence Between Physical Systems and RESim Simulations

F.1 Overview

This appendix identifies potential sources of divergence between RESim models and real world systems and presents some strategies for mitigating divergence. Models that lack fidelity with target hardware may lead to divergent execution of software. Consequences can range from scheduling differences, (which generally also diverge between boots of the same hardware), to substantially different behavior between the model and the physical system. For example, on some boots of a simulation, a set of Ethernet devices may be assigned incorrect names, e.g., "eth0" is assigned to the device that should be "eth1". In some cases, these divergences can be mitigated if detected. In our example, if the eth0/eth1 are found to have been swapped, then reversing their corresponding connections to switches might mask the problem, restoring fidelity between the simulation and the physical system. (See F.4 below.)

F.2 Timing

Simics processor models execute all instructions in a single machine cycle. The quantity of machine cycles required to execute instructions varies by instruction on real processors. Most designs for concurrent process execution do not rely on cycle-based timing. However, race conditions that appear on real systems may manifest differently on simulated systems.

F.3 Model Limitations

It is not always practical to obtain a high fidelity model of every component in a target system. TBD: Explore model building, e.g., to simulate an fpga accessed via a PCI bus device.

F.4 Diddling – dynamic modifications to memory and topology

RESim includes *diddling* functions that dynamically modify modeled elements and connections, triggered by system events. The `runToDiddle` function triggers on the writing of a specified string (or regular expression) via the `write` system call. Note in all the subfunctions listed below, the trigger string identifies the write operation that triggers the action. The `runToDiddle` function has a parameter that names a file containing diddle directives. The format of directives files depend on the subfunction as listed below.

- `sub_replace`– Replace a substring within a write buffer with a given string. The directives file includes one or more sets of directives. An example directives file looks like:

  ```
  sub_replace
  #
  # match
  # was
  # becomes
  root:x:0:0:root
  root:x:
  root::
  ```

  This example might be run when the `su` command is captured in the debugger.

- `full_replace` – Replace the entire write buffer with a given string. The directives file includes a single directive whose replacement string may include multiple lines.

```
full_replace
KERNEL=="eth*", NAME="eth
SUBSYSTEM=="net", ACTION=="add", DRIVERS=="?*", ATTR{address}=="00:e0:27:0f:ca:a8", \
  ATTR{dev_id}=="0x0", ATTR{type}=="1", KERNEL=="eth*", NAME="eth0"

# PCI device 0x8086:0x1001 (e1000e)
SUBSYSTEM=="net", ACTION=="add", DRIVERS=="?*", ATTR{address}=="00:e0:27:0f:ca:a9", \
  ATTR{dev_id}=="0x0", ATTR{type}=="1", KERNEL=="eth*", NAME="eth1"
```

- `match_cmd` Execute a list of Simics commands when the trigger string is found in a write buffer and a separate substring is also found. If the trigger string is found, the function will terminate, i.e., no more `write` syscalls will be evaluated.

```
match_cmd
#
# match (regx)
# was (regx)
# cmd
KERNEL=="eth\*", NAME="eth
("00:e0:27:0f:ca:a8".*eth1|"00:e0:27:0f:ca:a9".*eth0)
disconnect $VDR_eth0 switch0.device1
disconnect $VDR_eth1 switch1.device1
connect $VDR_eth0 cnt1 = (switch1.get-free-connector)
connect $VDR_eth1 cnt1 = (switch0.get-free-connector)
```

G What is different from Simics?

RESim is based the Simics simulator, including the *Hindsight* product. It does not incorporate the *Analyzer* product OS-Awareness funtions. RESim includes its own OS-Awareness functions. The Simics Analyzer features are primarily intended to aid understanding and debugging of *known* software, i.e., programs for which you have source code. RESim is intended to reverse engineer unknown software, and thus does not assume possession of source code or specifications.