Reverse Engineering Networked Systems with RESim

External dynamic analysis of heterogeneous networks of computers

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 you’d 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]](#footnote-2) 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 full 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 a disassembler/debugger that drives the running simulation. This disassembler/debugger, (REDis), allows setting breakpoints to pause the simulation at selected events in either future time, or past time. For example, REDis can direct the simulation state to reverse until the most recent modification of a selected memory address. ReDis allows the analyst to 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.

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.

# 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:
   1. Providence, i.e., which process created the process (or thread), and what programs were loaded via the execve system call.
   2. Files and pipes that had been opened (including those whose file descriptors were inherited from their parent), and those that are currently open.
   3. Linux socket functions, e.g. , connect, accept, bind, etc. Socket connect attempts to external components are highlighted, as are socket accepts from external components.
   4. 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].

# Dynamic analysis of programs executing in their environment

REDis couples an IDA Pro[[2]](#footnote-3) 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 process’s address space.

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.

# Limitations

RESim analyzes Linux-based systems for which copies 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 requires 32-bit X86 kernels, but will be adapted to support 64-bit kernels and applications. It can also be extended to support alternate architectures, e.g., ARM, supported by Simics processor models[[3]](#footnote-4).

# Work Flow

Simulated systems are defined within Simics configuration files that identify processors and interface devices, e.g., network cards, disks and system consoles. Once a system is modeled, it is loaded into the Simics platform and the disk image is booted.

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.

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 REDis disassembler/debugger for a specified process

When REDis 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.

REDis 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.
* 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

# Development and Availability

RESim is intended to be offered as a network service to users running local copies of IDA Pro (for REDis) 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. REDis functions are implemented using IDAPython.

# 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 zero 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.

**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.

# Appendix A: Introspection on a custom stripped kernel

Use of introspection, (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.

# Appendix 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.

# Appendix 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 REDis plug-in retrieves the address of the library and causes IDA to rebase using that offset.

# Appendix E: 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.)

1. Simics is a full system simulator sold by Intel/Wind River, which holds all relevant trademarks. [↑](#footnote-ref-2)
2. The Ida PRO disassembler and debugger is a product sold by Hex-Rays, which holds all relevant trademarks. [↑](#footnote-ref-3)
3. A summary of Linux device models is at: https://www.windriver.com/products/simics/simics-supported-targets.html [↑](#footnote-ref-4)