



RESim User's Guide

Reverse Engineering and vulnerability analysis of software on networks of heterogeneous computers by instrumenting simulated hardware

February 3, 2023

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 4 |
| 1.1 | This Guide | 4 |
| 1.2 | Analysis artifacts | 5 |
| 1.3 | Dynamic analysis of programs executing in their environment | 5 |
| 1.4 | Limitations | 6 |
| 1.5 | Development and Availability | 6 |
| 2 | Notional Workflow | 6 |
| 2.1 | Workspace directories | 7 |
| 2.2 | Configuration files | 7 |
| 2.3 | Kernel parameter extraction | 7 |
| 2.4 | Find interesting processes | 7 |
| 2.5 | Analyze a process | 8 |
| 2.6 | Code coverage with AFL | 8 |
| 3 | RESim commands | 8 |
| 3.1 | General display | 9 |
| 3.2 | Process tracing | 9 |
| 3.3 | Saving state | 9 |
| 3.4 | Process Analysis | 10 |
| 3.5 | Data tracking | 12 |
| 3.6 | Vulnerability detection | 13 |
| 3.7 | System modification | 13 |
| 3.8 | Code coverage | 14 |
| 3.9 | Fuzzing | 14 |
| 3.10 | Msc | 16 |
| 4 | Defining a target system | 16 |
| 4.1 | ENV section | 16 |
| 4.2 | Target sections | 17 |
| 4.3 | Target platforms | 18 |
| 4.4 | Network definitions | 19 |
| 4.4.1 | Configurting VLANs | 19 |
| 4.5 | Driver component | 20 |
| 5 | Running the simulation | 20 |
| 5.1 | Installation | 20 |
| 5.1.1 | Missing Simics platform files | 20 |
| 5.1.2 | IDA Pro installation | 21 |
| 5.2 | Getting started | 21 |
| 5.2.1 | Kernels with ASLR | 21 |
| 5.2.2 | Kernel Parameters for 32-bit compatibility | 22 |
| 5.3 | IDA Pro | 22 |
| 5.3.1 | RESim IDA windows | 22 |
| 5.3.2 | RESim IDA options | 23 |
| 5.4 | Ghidra | 23 |
| 5.5 | Dynamic modifications to memory and topology | 23 |
| 5.5.1 | Dynamic modifications to multiple computers | 25 |
| 5.5.2 | Dmods in the background | 25 |
| 5.5.3 | Manual modifications to topology | 25 |
| 5.6 | Sending data | 25 |
| 5.7 | Tracking data | 25 |
| 5.7.1 | Watch Marks | 26 |
| 5.7.2 | Tracking injected data | 26 |
| 5.8 | Preparing for data injection | 26 |
| 5.8.1 | Tracking data through pipes | 27 |
| 5.8.2 | Tracking libraries | 27 |
| 5.8.3 | When are you done? | 27 |
| 5.8.4 | Warning | 27 |

| | | |
|--|---------------------------------------|-----------|
| 5.8.5 | Tracking backwards | 27 |
| 5.8.6 | Removing unwanted traffic | 28 |
| 5.9 | Code coverage | 28 |
| 5.9.1 | Coverage from fuzzing | 28 |
| 5.9.2 | Ad-hoc code coverage | 28 |
| 5.9.3 | Branches not taken | 29 |
| 5.9.4 | BNTs from fuzzing | 29 |
| 5.9.5 | Hits in libraries | 30 |
| 5.10 | Selecting checkpoints | 30 |
| 6 | Fuzzing with AFL | 30 |
| 6.1 | Seeds | 31 |
| 6.2 | Multiple UDP packets | 32 |
| 6.3 | Fuzzing TCP | 32 |
| 6.3.1 | Partial reads | 32 |
| 6.3.2 | TCP timing and packetizing | 32 |
| 6.3.3 | State of the target system | 32 |
| 6.4 | Packet filters | 33 |
| 6.5 | Address jumpers | 33 |
| 6.6 | Fuzz another library | 33 |
| 6.7 | Fuzz another process | 33 |
| 6.8 | Thread isolation | 33 |
| 6.9 | Crash analysis | 34 |
| 6.10 | Update code coverage | 34 |
| 6.11 | Fuzzing performance | 34 |
| 6.12 | Why fuzz with full system simulation? | 34 |
| 6.13 | Parallel fuzzing | 35 |
| 6.13.1 | Fuzzing with multiple computers | 35 |
| 6.14 | False paths | 36 |
| 7 | Example workflows | 36 |
| 7.1 | Watch consumption of a UDP packet | 36 |
| 7.2 | Reverse engineer a service | 36 |
| 7.3 | Observe changes in outputs | 37 |
| 7.4 | Track buffer accesses | 37 |
| 7.5 | Find code divergence | 37 |
| 7.6 | Branches not taken | 37 |
| 7.7 | Packaged example with public images | 37 |
| 8 | Example targets | 37 |
| 8.1 | Cyber Grand Challenge Services | 37 |
| 9 | Implementation strategy | 38 |
| 10 | Troubleshooting | 38 |
| 11 | Data Stores | 39 |
| 12 | Utility scripts | 39 |
| 13 | Testing | 40 |
| Appendices | | 41 |
| Appendix A Analysis on a custom stripped kernel | | 41 |
| Appendix B Detecting SEGV on a stripped Linux Kernel | | 41 |
| B.1 | Faults on ARM | 41 |
| B.2 | In process | 41 |
| Appendix C External tracking of shared object libraries | | 42 |
| Appendix D Analysis of programs with crude timing loops | | 42 |

| | |
|--|-----------|
| Appendix E Breakpoints can be complicated: Real and virtual addresses | 42 |
| Appendix F Divergence Between Physical Systems and RESim Simulations | 43 |
| F.1 Overview | 43 |
| F.2 Timing | 43 |
| F.3 Model Limitations | 44 |
| F.3.1 VLANs | 44 |
| F.3.2 File descriptors | 44 |
| Appendix G What FD is this? | 44 |
| Appendix H Context management implementation notes | 44 |
| Appendix I What is different from Simics? | 45 |
| Appendix J IDA Pro issues and work arounds | 45 |
| Appendix K Simics issues and work arounds | 45 |
| Appendix L Performance tricks | 45 |
| Appendix M New disk images | 45 |
| Appendix N Implementation notes | 46 |
| Appendix O ToDo | 46 |
| O.1 Msc | 46 |
| O.2 I/O via threads | 46 |
| O.3 Tracing library calls | 47 |
| O.4 Watching process exit whilst jumping around time | 47 |
| O.5 Defining new targets | 47 |
| O.6 Fuzzing: too many crashes | 47 |
| O.7 Branches not taken | 47 |
| O.8 Skipping over kernel calls | 47 |
| O.9 Data Watch ntohl | 47 |
| O.10 More on kernel data buffers | 48 |
| O.11 Data watch ad-hoc copies | 48 |
| O.12 UDP without headers | 48 |
| O.13 TCP conversations and fuzzing | 48 |
| O.14 Fuzzing crash detection | 49 |
| Appendix P Driver platforms | 49 |
| P.1 Notes on updating the driver | 50 |
| Appendix Q Simics user notes | 50 |
| Appendix R Injecting to kernel buffers | 51 |
| Appendix S Troubleshooting | 51 |
| S.0.1 Missing syscalls in logs | 51 |
| S.1 Real networks: WARNING | 51 |
| S.2 Failed cat to /dev/udp/localhost | 52 |
| S.3 Real networks and UDP | 52 |
| S.4 Backtracing malloc'd addresses | 52 |
| S.5 Fork exit | 52 |
| S.6 Fuzzing | 52 |
| S.7 Fuzzing TCP | 52 |
| S.8 Multipacket workflows | 52 |
| S.9 Your target process is not scheduled in the snapshot | 53 |

1 Introduction

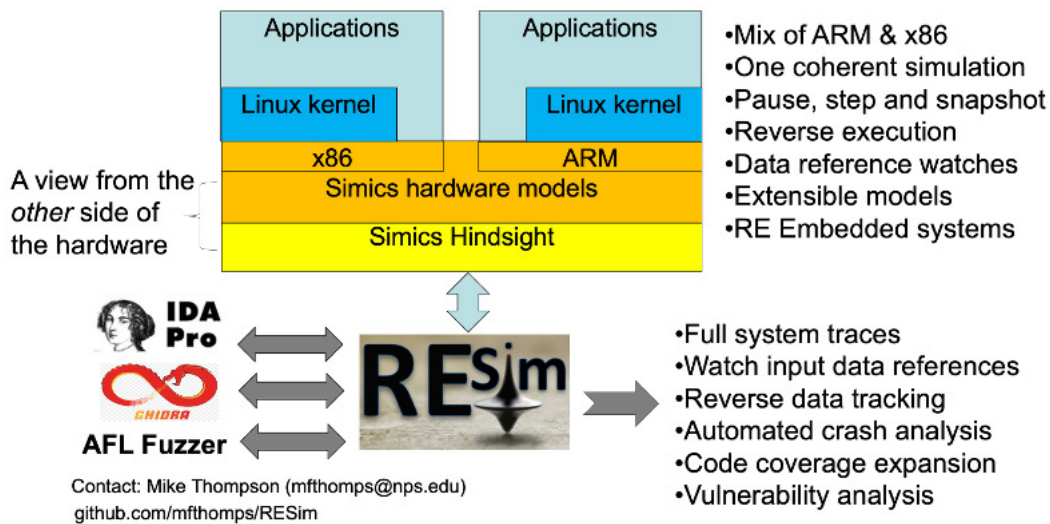
Imagine you would like to dynamically analyze, (e.g., perform cyber testing on), the processes on embedded computers within a network, and assume you'd like to perform this analysis without altering the systems to add instrumentation and without having a shell on the systems. RESim is a dynamic software analysis tool that provides detailed insight into processes, programs and data flow within networked computers. RESim simulates networks of computers through use of the Simics¹ 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. Insight into software behavior is obtained by instrumenting the simulated hardware rather than instrumenting the software.

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 recorded through dynamic analysis of a running simulated system without installation or injection of software into the simulated system, and without detailed knowledge of the kernel hosting the processes. The 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 explicitly modify memory or register content, and can also dynamically augment memory based on system events, e.g., change a password file entry as it is read by the `su` program (see the `dmod` function described in 5.5).

RESim also provides interactive analysis of individual executing programs through use of either the IDA Pro or the Ghidra 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. During a RESim session, any point within the simulation can be *bookmarked*, and that execution state can later be restored.

The *American Fuzzing Lop* (AFL) fuzzer is integrated with RESim, which injects fuzzed data generated by AFL directly into simulated memory. Instead of constructing custom test harnesses for each target program, RESim enables fuzzing of programs as they execute in their native environments, which may include substantial interaction with other processes or computers prior to reaching the state at which fuzzing is to commence. RESim creates a memory-based snapshot of that state, and returns the system to that state after each fuzzing iteration.

The analyst can generate reloadable checkpoints at any point during system execution, and these checkpoints can then be shared with other analysts and used as the starting point of future RESim sessions. These checkpoints include the full target system state (e.g., similar to a VM snapshot) as well as RESim context information such as information about currently running processes.



1.1 This Guide

The remainder of this introduction provides an overview of RESim features and its limitations and availability. Following the introduction:

- Section 2 describes a notional workflow, highlighting functions and features of RESim and how an analyst might use them to reverse engineer a network of computers.

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

- The set of commands supported by RESim are listed in section 3.
- Section 4 details the elements of RESim configuration files (ini files) that define the system to be simulated.
- The mechanics of installing and running RESim are described in section 5. This includes descriptions of data tracking (forward and reverse) and code coverage.
- Example workflows for RESim are provided in section 7.
- Example disk images and targets are listed in section 8.
- Use of AFL to expand code coverage and identify vulnerabilities is described in section 6.
- Section 9 describes the implementation strategy and some of its consequences.
- Section 10 provides some trouble-shooting tips.
- The Appendices provide a set of notes and hints that have not yet been integrated into the body of the guide.

1.2 Analysis artifacts

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

1. A 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].

Artifacts associated with individual processes (and their associated threads) include:

- Maps of shared object libraries loaded by each thread, including their load addresses
- Records of references to input data, including copies of such data, as the target program consumes the data. These *Watch Marks*, (see 5.7), can be dynamically loaded to skip the simulation to the point of the reference.
- Reverse data tracks that trace sources of memory or register values in terms of exchanges between memory and registers, potentially leading back to initial ingest of the data, e.g., via a `recv` system call.
- Data written to selected files or file descriptors, (see the `traceFile/traceFD` commands).
- System traces as previously described, but constrained to actions taken by threads within the process being analyzed.

1.3 Dynamic analysis of programs executing in their environment

RESim couples an IDA Pro 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.

Analysis is performed entirely through external 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 observation functions do not affect the state of the simulated system. For example, when viewing code with the IDA Pro or Ghidra debugger, addresses that are not yet paged in will appear as containing zeros. In most other systems, the debugger or its agent would run on the target, and thus the mere act of viewing the address with a debugger would cause the kernel to page in the memory containing the referenced code.²

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.

1.4 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 models³. RESim is currently limited to single-processor (single core) models. Simics supports multi-processor simulations (at reduced performance), but RESim has not yet been extended to monitor those. See section F for information on potential divergence of simulations from real systems.

1.5 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. See the *RESim Remote Access Guide*. 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.

- Simics is available as a commercial product from Wind River. A free version for Intel processors is available at <https://www.intel.com/content/www/us/en/developer/articles/tool/simics-simulator.html>
- IDA Pro extensions are implemented using IDAPython, and are included within the RESim repo at <https://github.com/mfthomps/RESim/simics/ida>
- The Ghidra plugins for RESim are available at: <https://github.com/mfthomps/RESimGhidraPlugins> A fork of gdb needed for use with the Ghidra plugin is at <https://github.com/mfthomps/binutils-gdb>.
- A fork of the AFL fuzzer integrated with RESim is available at: <https://github.com/mfthomps/AFL>.

2 Notional Workflow

This section provides an overview of how RESim can be used by an analyst to reverse engineer a system. Details are provided elsewhere in this guide. The general steps include:

- Extract software images from target systems.
- Identify Simics models to simulate the target hardware.
- Create a RESim workspace directory with which to run the simulation
- Construct a RESim configuration file to identify system image file locations and RESim parameters.
- Use RESim to extract a set of kernel parameters if not already extracted for the kernel images being used.
- Run the simulation to produce full system traces and processing reports for the simulated systems.
- Identify processes of interest, and use the RESim IDA Pro debugging client to analyze their behavior, e.g, the protocols they consume.
- Use the integrated AFL fuzzer to improve code coverage within the targeted process.

²See E for another example of the implications

³A summary of Simics device models is at: <https://www.windriver.com/products/simics/simics-supported-targets.html>

- Analyze RESim sessions to identify unexplored code paths and identify input data to reach those paths, potentially feeding those back to AFL.
- Automatically assess any crashes generated by AFL to identify potentially exploitable vulnerabilities.
- Use RESim to analyzing the vulnerabilities and craft proof of concept exploits.

2.1 Workspace directories

Simics simulations are run from workspace directories prepared using the `resim-ws.sh` command, which creates a Simics workspace within a new empty directory and populates it with selected RESim files. The workspace is where simulation-specific configuration files and scripts are to reside. Snapshot data is also stored within workspace directories. You may share snapshots between workspaces by using symbolic links.

2.2 Configuration files

Simulated systems are defined within RESim configuration files (see [4](#) that parameterize pre-defined Simics scripts to identify processors and interface devices, e.g., network cards, disks and system consoles. RESim currently includes the following platforms:

- A general purpose X86 platform with disk, multiple ethernet and serial ports;
- A generic ARM Cortex A9 platform with disk, multiple ethernet and serial ports.
- An ARMV5 platform based on the ARM926EJ-S processor. This is a partial implementation of the ARM Integrator-CP board. It currently only supports the initial RAM disk, one ethernet and serial ports.

Other platforms can be modeled via Simics, the details 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.

2.3 Kernel parameter extraction

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 `CREATE.RESIM.PARAMS` directive within the RESim configuration file directs the tool to automatically analyze the running kernel and extract the desired parameters. This allows RESim to analyze disparate Linux kernels without a priori knowledge of their versions or configurations ⁴. See section [5.2](#).

2.4 Find interesting processes

Once the kernel parameters have been extracted, the RESim configuration file is modified to reference the parameter file, and the simulation is restarted. The user is presented 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
- Enable reverse execution, i.e., allow reversing to events from that point forward
- Set and run to breakpoints, either in the future or in the past.
- Target RESim to focus on a specific process thread group, e.g., for dynamic analysis using IDA Pro.

⁴Not to be confused with the similar function included with Simics Analyzer product. RESim uses an alternate strategy for OS-Awareness.

See section 3 for details of RESim commands.

A typical strategy with RESim is to initially perform a full system trace on a system as a means of identifying programs of interest, e.g., which can then be further analyzed using the interactive IDA Pro client. The `traceAll` command described below in 3 generates such a trace. Note that a typical Linux-based system can perform tens of thousands of system calls during initial boot processing. It may save considerable time to use the `toProc` command to run forward to some event such as creation of `rsyslogd` before issuing the `traceAll` command. Note that `toProc` tracks processes creation and some system configuration settings such as IP addresses set using `ip` or `ifconfig` commands, which can be seen using the `showNets` command. Use the `writeConfig` command to create a checkpoint, and then update your ini file to begin at that checkpoint using the `RUN_FROM_SNAP` directive. Trace files are created in the `/tmp` directory. During a trace, if you use the Simics stop and run commands, the tracing will continue. While stopped, you may use the `tasks` commands to see which processes are currently running. Or the `showBinders` command to see network ports being listened to.

2.5 Analyze a 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.
- Track references to data read in via a given FD, as well as references to copies of that data, e.g., via `mempcpy`.
- 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

2.6 Code coverage with AFL

Using RESim to understand the protocols handled by a process, (and associated vulnerabilities), typically involves providing the process with crafted data and observing its behavior and modifying data to achieve greater code coverage. AFL contributes to code coverage expansion and vulnerability identification through automated guided fuzzing. AFL provide RESim with randomized inputs that it continually augments based on feedback that RESim generates reflecting code paths reached by each input.

A set of RESim functions can then be used to replay AFL sessions to automatically identify and display unexplored code paths within the IDA client. This allows the analyst to identify and generate inputs for additional code paths, and feed these inputs back into new AFL sessions. See section 6 for information on the use of AFL.

See 7 for detailed examples of RESim workflows.

3 RESim commands

The following RESim commands are issued at the Simics command prompt, naming the commands as methods of the `cgc` python module, e.g., "`@cgc.tasks()`". Interfaces for all of these commands are implemented in the python script at `simics/monitorCore/genMonitor.py`. If this documentation falls out of sync, or you need to fix broken stuff, refer to that file (and the dependent python classes, which are all in that directory.)

There are also a set of utility commands that run from a bash prompt (rather than from Simics). See section 12 for a list of those.

3.1 General display

- **show** – Show current process information for the currently monitored target.
- **tasks** – List currently executing process names and their PIDs for the currently monitored target.
- **showThreads** – List threads of process currently selected for analysis, e.g., via **debugProc**.
- **showTargets** – List the target cells that may be monitored.
- **setTarget** – Set the currently monitored target.

3.2 Process tracing

- **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. See the scripts in **RESim/postscripts** to parse system call logs and create reports on file, network and IPC (System V) usage. To limit tracing to the current thread, first issue the **debugThis** command.
- **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. **Note** Generally use **traceAll** instead of this command. This command has some value in that it results in breakpoints on calculated entry points vs **sysenter** type entry points. This may aid in some **RESim** debugging. See the scripts in **RESim/postscripts** to parse system call logs and create reports on file, network and IPC (System V) usage.
- **saveTraces** – Combines the next 4 items.
- **showProcTrace** Generate a process family summary of all processes that executed since the **traceProcess** (or **traceAll**) command.
- **showNets** Display network configuration commands (e.g., **ifconfig** 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.
- **showConnectors** Display programs that use **connect** to open sockets intended for use during process tracing to identify processes that connect to 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)]**. The **trackIO** command can be directed to include traced file output in the watch marks, e.g., to see where log messages occur within the time line of watch marks.
- **traceFD(FD)** Copy all writes that occur to a given FD, e.g., **stdout**. Output is in **/tmp/output-fd-[FD].log** Assumes **traceAll** is set.
- **flushTrace** - Flush trace output to the trace log files.
- **toProc** 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.
- **autoMaze** – Avoid being prompted when tracing detects a crude timing loop or other events that are repeated many times in a loop. See section [D](#).
- **instructTrace(file, all_proc=False, kernel=False, watch_threads=False)** -- Generate an instruction trace and save it into the named file. Use the **kernel** flag to observe traces within the kernel. Use **watch_threads** to trace all threads of the current process.

3.3 Saving state

- **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, shared object maps and network configuration commands for reference subsequent to use of the read-configuration function.
- Also see **prepInject**.

3.4 Process Analysis

These functions support interactive analysis of the threads of a process, e.g., to run until some system call is made. Also see the *Tracking data* functions in the following subsection. Analysis functions depend on data gathered while the target process is being loaded and initialized. In general, you must use the `debugProc` function to capture that information. You may then create a checkpoint. Subsequent sessions started from that checkpoint, or subsequent checkpoints, can then use alternate functions to select debugging, e.g., the `debugPidGroup` function. RESim gathers information on all processes started while processing the `debugProc` command, so it is possible to create checkpoints from which multiple processes can be selectively debugged. To put this another way, you cannot choose to debug some process that is already running on the simulated system unless you took steps to gather its data prior to the process being loaded.

- `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. Most of the commands listed below have analogs available from within IDA Pro, once the RESim `rev.py` plugin is loaded.

If execution transfers to a shared object library of interest, the associated library file can be found via the `getSOFile` command described below. Load that file into IDA Pro and rebase to the SO address found via `showSOMap` prior to attaching the debugger. If you have run `reTrack` or `injectIO`, you must re-run the command in order for the Watch Marks to detect and report mem operations, e.g., `memcpy` – and then refresh the IDA Data Watch window.

If you prefix the given process name with `sh` , (sh followed by a space), RESim will look for a shell invocation of the script name that follows the `sh`.

- `debugPidGroup(pid)` – Initiate the debugger server for a given PID. Intended for use after starting a session from a checkpoint created using `writeConfig`. This assumes process information had been generated in a previous session that was saved.
- `debugSnap` – Assumes the current session was loaded from a snapshot, it use `debugPidGroup` to debug the process being debugged when the snapshot was made. **NOTE** If you use `debugSnap` after running for a while, you may well lose process state, specifically system call information about threads that are waiting in the kernel. If you want to run ahead to let the system settle out, first use `debugSnap` and then use `noReverse` to improve speed.
- `debugIfNot` – Will call `debugPidGroup` for the currently scheduled process.
- `debugThis` – Limit debugging to the current thread, ignoring other threads.
- `getStackTrace` shows the call stack as seen by the monitor. The Ida client uses this to maintain its view of the callstack. The monitor uses the IDA-generated function database (`.fun` files stored with the `.idb` files to aid in determining if potential instruction calls are to functions. The monitor-local `stackTrace` command displays a stack trace that uses the IDA function database to resolve names. (TBD, does not yet handle `plt`, and thus shows call addresses for such calls). This function is not always reliable, e.g., phantom frames may appear based on calls that occurred previously. ARM stack frames may be difficult to determine due to its myriad ways of calling and returning.
- `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.
- `revToSyscall()` – Reverse to previous syscall made by the current process.
- `runToConnect(search pattern)` Continue execution until a socket connection to an address matching the given search pattern.
- `runToBind(search pattern)` Continue execution until a socket bind to an address matching the given search pattern. Alternately, providing just a port number will be translated to the pattern `.*:N` where N is the port number.

- **runToAccept(FD)** Continue execution until a return from a socket accept to the given file descriptor.
- **runToIO(fd, nth=1)** Continue execution until a read, write, select, ioctl, accept or close with the given file descriptor. If nth is greater than 1, will run until the nth recv call.
- **runToInput(fd)** – Like runToIO, but only finds reads, receives, etc.
- **runToOpen(file)** – Stops at the open of a given filename.
- **runToWrite(substring)** – Stops when a process writes the given substring.
- **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. Also useful when you get lucky and execution is down a nop sled. This function can be slow, e.g., if there is a kernel system call between you and the text segment.
- **showSOMap(pid)** Display the map of shared object library files to their load addresses for the given pid (along with the main text segment).
- **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. Note that ROP may throw this off, causing you to land at the earliest recorded bookmark. Use **revInto** to reverse following a ROP.
- **uncall** Reverse execution until the call instruction that entered the current function. This may be unreliable with some ARM programs.
- **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.
- **toPid(pid)** – Run forward until the given pid is scheduled. Use -1 to indicate any of the threads being debugged.
- **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. The **listBookmarks** command will list bookmarks, displaying index numbers. These numbers can be provided to the **goToDebugBookmark** command instead of the bookmark string. Note these bookmarks are separate from the data watch bookmarks, which can be viewed using the **showWatchMarks** command.
- **runToKnown** Continue execution until a text range known to the SOMap (see **showSOMap()**). Intended for use if execution stops in got/plt or other loader goo.
- **runToOther** Continue execution until a text range known to the SOMap (see **showSOMap()**) – but not the current text range – is entered. Useful if you are in some library called by some other library, and you want to return to the latter.
- **doBreak** Sets a breakpoint on a given address. This command can be useful when wishing to break while running other commands. For example, using **doBreak** prior to running **injectIO** may be more reliable than simply setting a Simics break and it will clean up other breaks and haps that might interfere with your subsequent analysis.

3.5 Data tracking

Also see [5.7](#)

- **trackIO(FD, reset=False, count=1, mark_logs=False, kbuf=False)** – Combines the **runToIO** and the **watchData** functions to generate a list of data watch bookmarks that indicate execution points of relevant IO and references to received data. This list of data watch bookmarks is displayed using **showWatchMarks** or in the IDA client **data watch** window (right click and refresh). If an **accept** call with the given FD is detected, the FD being tracked will be changed to that returned by the accept call. The **trackIO** function will break simulation after **BACK_STOP_CYCLES** with no data references. If the debugged processes are in the kernel waiting to read on the given FD, RESim will ensure that call is tracked. The **count** parameter can be used to track the Nth read or recv. Data watches persist after the call, e.g., to support **retrack** described below. Each use of **trackIO** will reset all data watches, but does not clear watch marks, i.e., you can still skip to those simulation cycles. Also see the **tagIterator** command. The **reset** parameter will cause the reverse time origin to be reset if the kernel was waiting on a read/recv. See [S.1](#) Data to be consumed by the target can originate from a simulated computer, e.g., a driver, or via a real network using something like:

```
cat foo.io > /dev/udp/127.0.0.1/6060
```

Note that sending multiple packets via real networks will cause the origin to reset on each packet, thereby limiting the range of reverse execution. It is generally simpler to use a driver computer to send multiple packets, using a program such as **simics/bin/drive-driver.py**. The **-d** option of that program executes the Simics magic instruction to cause RESim to reset its origin just prior to the initial data transmission. This also causes RESim to disconnect the real network to ensure that the reversible range does not receive any real-world leakage.

Use of the **mark_logs** option will include corresponding log entries from any use of **traceFile** as watch marks. This may help display diagnostics within your watch mark list as parsing encounters errors.

The **kbuf** option is used in conjunction with **prepInjectWatch** to record kernel buffers used when receiving the data. This data must be populated with a designated character, currently “Z”. And there must be as much data as you expect to read during **injectIO** sessions. The designated characters are used by RESim to identify the end of individual kernel buffers, and thus are critical to determining the size and location of the different kernel buffers. It is also critical that the data be of a form that will be read by the application because RESim locates kernel buffers by backtracing data read into application buffers. However, if you find the first kernel buffer is as large as any data you might send, then a single application read will suffice. If AFL filters are to be used, e.g., to generate CRCs, then apply the filter before sending the data to the target via the driver. Take care to avoid extraneous characters, e.g., newlines at the end of the test data.

- **trackProgArgs** Track references to arguments provided to the program as argv values.
- **trackCGIArgs** Track references to arguments provided to the program as packaged by cgi-bin.
- **retrack** – Intended for use after modifying content of an input buffer in memory. This will track accesses to the input buffer. Note that this function does record additional IO operations, BUT WILL reflect subsequent access to the existing watch buffers. (TBD, terminate on access to remembered FD?)
- **trackFile** – Currently works only with files opened by **xmlParseFile**. The function notes all memory malloc’d within **xmlParseFile** and then tracks its access, e.g., via **xmlGetParam**, adding watch marks as it goes.
- **showWatchMarks(old=False)** Display a list of watch marks created by **watchData** or **trackIO**. Use the **old** flag to view stale watch marks, i.e., those that occurred prior to the origin (which gets reset on each data injection.)
- **goToDataMark(watch_mark)** Skip to the simulation cycle associated with the given **watch_mark**, which is an index into the list of data watch bookmarks generated by **trackIO** or **injectIO**.
- **getWatchMarks()** Return a json list of watch marks created by **watchData** or **trackIO**, used by the IDA client.
- **tagIterator(index)** Tag a watch mark as being an iterator. The associated function is added to a file stored along with IDA functions. The intent is to avoid data watch events on each move, or access, e.g., a crc generator.

- **trackFunctionWrite(fun)** Record all write operations that occur between entry and return from the named function.
- **goToWriteMark(write_mark)** Skip to the simulation cycle associated with the given `write_mark`, which is an index into the list of write watch bookmarks generated by `trackFunctionWrite`.
- **getWriteMarks()** return a json list of write marks created by `trackFunctionWrite`.
- **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
- **injectIO(iofile, stay=False, n=1, target=None, targetFD=None, cover=False, break_on=None, trace_all=False, mark_logs=False, no_reset=False)** – Assumes you have previously used `prepInject` to create a snapshot (see 5.8 and that snapshot has been loaded. The content of the given `iofile` is written into the read buffer, and the register reflecting the count is modified to reflect the size of the file. If `stay` is False, the `retrack` function is then invoked. If `keep_size` is set, then the size register will not be altered, which is intended for use when replaying files trimmed by AFL. This `injectIO` function intended use is to rapidly observe execution paths for variations in input data. The command will automatically run the `debugPidGroup` command on the current PID. The optional `n` parameter causes the data file to be treated as `n` different packets. The optional `target` and `targetFD` parameters name a different process and its FD whose data references are to be tracked. For example, if the original receiving process does some processing and sends derivative data to a pipe, the target could be the process at the read side of the pipe. If the `cover` parameter is true, code coverage will be tracked and saved in a hits file. This may take a while – wait for the Simics prompt to display a message such as `Coverage saveCoverage to....` The `break_on` will break on execution of the given address.

The `trace_all` option will generate a system call trace instead of watch marks.

Use `mark_logs` to include log entries generated using `traceFile` or `traceFD` within the watch marks.

Use `no_reset` to stop the simulation rather than resetting the origin, e.g., to adjust a return value from a read.

- **traceInject(iofile)** – (DEPRECATED) See the `trace_all` options to `injectIO` and `playAFL`.
- **traceMalloc** – track calls to the `malloc` and `free` functions and includes those events in the list of watch marks. Use `showMalloc` to then list by pid, block address and size.
- **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. List data watches using `showDataWatch`. Note however, that data watches are based on the `len` field given in the read or `recv`, and thus data references are not necessarily to data actually read (e.g., a read of ten bytes that returns one byte would break on a reference to the fifth byte in the buffer.)

3.6 Vulnerability detection

- **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 [B](#) for information about what we mean by `SEGV`, and how we catch it.
- **watchROP** – Watch for return instructions that do not seem to follow calls. This is available while debugging a process.

3.7 System modification

- **writeReg** Write value to a register (Note: deletes existing bookmarks.)
- **writeWord** Write word to an address. This function will delete existing bookmarks and create a new origin. If there are data watch marks, i.e., from a `trackIO`, these are deleted except for any that are equal to the current cycle. The upshot of this is that if you want to modify memory and then rerun a `trackIO`, do so while at the first datamark.

- **writeString** - Write a string to an address. Use double escapes, e.g., two backslashes and an n for a newline. (Note: deletes existing bookmarks.)
- **runToDmod()** Run until a specified read or write system call is encountered, and modify system memory such that the result of the read or write is augmented by a named Dmod directive file. See 5.5
- **modFunction(fun, offset, word)** Write the given word at an offset from the start of the named function. Intended for use in setting return values, e.g., force eax to zero upon return. Bring your own machine code. TBD accept an assembly statement?
- **showLinks** - Display symbolic names of network connections. These may then be used in DMOD directives, or manually using Simics **connect** and **disconnect** commands.
- **jumper(from, to)** - Alter control flow by causing execution to jump to a given address whenever a given from address is hit. See the **EXECUTION_JUMPERS** ini environment value.

3.8 Code coverage

- **mapCoverage** - Set breakpoints on all basic blocks in the text segment and use them to track code coverage. The basic block coverage is saved in a <prog>.hits file in IDA db directory, and will be read by the IDA client when the **color** option is used. Subsequent uses of **mapCoverage** will reset the coverage tracking. Use **stopCoverage** to stop basic block tracking. The basic blocks database is read from the .blocks file created by the IDA client findBlocks script. See section 5.9 for information on how the IDA client represents code coverage and highlights branches that have not been taken.
- **showCoverage** - display summary of basic block coverage. Also see the IDA client **color** and **clear** options.
- **goToBasicBlock** - Skip the simulation state to the first hit of the block named by an address (requires use of **mapCoverage**..)
- **addJumper(from, to)** - Define a jumper to cause execution to skip from a from block to a to block.

3.9 Fuzzing

- **afl(fname=None, target=None, targetFD=None, dead=False)** - Typically the **runAFL** utility (from a bash shell in your workspace directory) is used instead of this command. Uses a network connection to communicate AFL server github.com/mfthomps/AFL. The simulation is assumed to be loaded from a checkpoint created using **prepInject**. Provide **fname** to name a library, i.e., if the program's main text blocks is not what is to be instrumented. Use **target** and **targetFD** to name some other process and the FD from which it reads if the fuzzing target is other than the consumer of the AFL-generated data. The **dead** option creates a filter to avoid instrumenting basic blocks hit by other threads. Use **aflTCP** for TCP sessions, though note these assume all data is read into the same input buffer. See section 6 for details.
- **prepInject(fd, configfile, count=1)** - Prepares a simulation for injectIO or fuzzing by running to an input system call for the given FD. It saves the post-call state in a snapshot file with the given name. Saved state includes the syscall call and return instruction addresses for use in multi-packet UDP fuzzing, and buffer address and size. Note using this with TCP and real networks can lead to undefined behavior, and thus you should use a driver computer instead of real networks. Use the count to stop the simulation subsequent to the nth input system call, e.g., to get an initial state for fuzzing that is predicated on a preamble exchange. This may require use of the **simics-magic** program on a driver computer. See 5.8 for details.

Note that the data you provide for **prepInject** should not include multiple UDP packets because you want the **recv** to not see more data after consuming your subsequent data injections.

Consider providing **prepInject** with largish amounts of data to ensure buffers that may cross page boundaries are populated.

- **prepInjectWatch(watchmark, configfile)** - For use when preparing to inject data directly into kernel buffers. Similar to **prepInject** but takes the index of a watch mark that can be a call to **ioctl**, or a call to **read**/**recv**. Generate the data marks by using **trackIO**, and providing the maximum amount of data you expect to read from kernel buffers. Use of the resulting snapshot with AFL or injectIO will cause the data to be injected into a kernel buffer. If the watch mark was an **ioctl** call, the kernel's buffer pointer values will be adjusted to reflect the new data size. Note this command does not support running forward past

the application references to the input data, e.g., the kernel may detect corruption on the `recv` following the end of the current data. Do not try to run these checkpoints forward to establish new checkpoints. It is important that received data all appears within the kernel buffer before the application makes its `recv` call. Otherwise, `injectIO` and AFL sessions will have kernel buffers overwritten by data on the wire. Consider using `drive-driver.sh` command.

- **prepInjectAddr(addr, configfile)** – For use when preparing to inject data directly into arbitrary application memory, e.g., when fuzzing `argv` parameters to a program. Consider priming the `prepInject` state with enough data to cross any page boundaries that might be part of the buffer.
- **fuzz(IOFile)** – Deprecated, AFL trims data itself just fine. Trim a data file to the minimum needed to not affect coverage. Assume the simulation is at a return from a read (e.g., using `prepInject`), set basic block coverage and iteratively execute while reducing the `IOFile` data size (padding with nulls) until a minimum is found. The final file in `/tmp/trimmer` is truncated and may need to be padded to be properly consumed, e.g., if the program insists on reading a minimum number of bytes. Intended for use in preparing minimal seed files for AFL.
- **playAFL(target, afl_mode=False, trace_all=False, fname=None)** – Play data input files discovered by AFL. Assumes `target` is a subdirectory of an output directory relative to the `AFL_DATA` environment variable). Each file in the `target`'s `queue` subdirectory(ies) will be played with code coverage tracked with new hits files generated in the `RESIM_IDS.DATA` directory. **Note:** While this command is running, you may see a static `simics>` prompt. That does not mean that Simics is not running – it is an artifact of the implementation. When running the command, you'll know it has finished when it displays where it stored the hits file. Tail the log if you suspect it has hung. Provide a library name, e.g., `libc.so.6`, to record coverage of a library. Note if `afl_mode` is set, no hits file is generated.

If the given `target` is a file, then only that file will be played. The `afl_mode` switch attempts to reproduce how the file was played in AFL, e.g., without temporary breakpoints and with the AFL backstop cycles. The `trace_all` switch will generate a system call trace. These are intended for use in discovering why AFL behaves in unexpected ways, e.g., lots of hangs, long sessions.

The `playAFL` command also generates a hits file for each AFL queue file and those are stored in a `coverage` directory alongside the `queue` directory. Those individual coverage files are intended to be read by the `findBB.py` utility to find sessions that lead to BNTs. Use `playAFLTCP` for tcp sessions. If `target` is a full path, then only that files will be played. **Note** AFL filters are not applied to replays, and thus you may see divergence from what AFL observed. The `runPlay` stand-alone command launches parallel instances of `RESim` running the `playAFL`, for use in replaying sessions created by parallel fuzzing. Note however this does not generate a new total hits file, it only creates coverage files.

- **crashReport(target)** Generate crash reports for each crash discovered by AFL. The given `target` is as defined in the `playAFL` command, except you can optionally provide the full path to a file, in which case only that file will be reported on. **NOTE:** when handling multi-packet data sets, use the `crashReport.py` utility (see section 12 to launch `RESim` multiple times, one for each data file. The reports are written to `/tmp/crash_reports`.
- **replayAFL(target, index, FD, [instance=N], cover=False, afl_mode=False)** – Replay a specific AFL session named by a target and index identifier, e.g., `0012`. Use the optional `instance` if parallel AFL is used. This function assumes a driver component. The driver will use the `sendDriver.sh` script to send the named file and a client program to the target using an address and port per the ini file `TARGET_IP` and `TARGET_PORT` values. For UDP sessions, `RESim` will use the `trackIO` function on the given `FD`. Use this to analyze multipacket processing. For TCP, use `replayAFLTCP` and the `FD` will be used to catch an `accept` system call. `RESim` will then use the returned `FD` for `trackIO`. Set `cover` to `True` to generate a coverage hits file. You can then start IDA with the `color` option, which will read that hits file. You can then double click on entries in the IDA BNT window to skip execution to the first hit of the selected BNT source block. Set `afl_mode` to report on blocks hit over 256 times in a session.

Note that function uses a driver to send data, and thus it should not be used with a `prepInject` snapshot.

- **aflInject(target, index)** – Replay an AFL session, similar to `replayAFL`, except the `injectIO` function is used without a driver component. Thus, only single-packet processing can be analyzed. Use `aflInjectTCP` for TCP sessions.
- **trackAFL(target)** – Use `injectIO` to replay all AFL queue files for a given target, and store the set of resulting watch marks as json files in the AFL output directory. Only the first packet is tracked. If the AFL sessions include multiple packets, then use the `runTrack` utility to start new Simics sessions for each queue file.

- `injectToBB(bb, fname)` – Find an AFL input file that leads to the given BB and play it using `injectIO`, breaking at the BB. The target is assumed to be the current RESim workspace directory name. Note the simulation breaks at the BB, and thus the watch marks do not reflect subsequent processing – it they will not reflect any watch mark in the target BB. (TBD, modify to process/track everything, and then skip back to the first hit of the BB.) Provide an `fname` if the `bb` is in a lib. The `bb` value is per the program addresses, and RESim will adjust it to reflect the lib load address.
- Also see utilities listed in section 12, e.g., to find branches not taken and AFL sessions that reach a given basic block.

3.10 Msc

- `runToCycle` – Run to an absolute cpu cycle on the current cpu. Useful for moving simulation forward to prepare to debug some process that is not created until after much processing. Cycles reported in syscall logs can help get you close to the point in time.
- `satisfyCondition` – (experimental) Assess the comparison instruction at a given address and attempt to satisfy it by altering input data. Initially handles simple ARM `cmp reg, <value>` instructions. The reverse track function is used to determine the source of the register content, and if that is a receive, it alters the data to satisfy the comparison and then uses `retrack` to run forward and track the new execution flow.
- `idaMessage` – display the most recent message made available to IDA. For example, after a `runToBind`, this will display the FD that was bound to.
- `setTarget` – Select which simulated component to observe and affect with subsequently issued commands. Target names are as defined in the RESim configuration file used to start the session.
- `saveMemory(addr, size, fname)` Write a byte array of the given size read from the given `addr` into a file with the given name.

4 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: an `ENV` section and one section per computer that will be part of the simulation. An example RESim configuration file is at:

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

A full example target is available as described in section 8.

4.1 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.
- `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
- `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.
- `DRIVER_WAIT` Causes RESim delay boot of target platforms (i.e., those other than the driver) until the user runs the `@resim.go()` command. Intended to allow you (or scripts) to configure the driver platform after it boots, but before other platforms will boot.
- `BACK_STOP_CYCLES` Limits how far ahead a simulation will run after the last data watch event, or the last code coverage hit.
- `HANG_CYCLES` Limits how many cycles a simulation will run under AFL until it is considered hung.
- `MONITOR` If set to `NO`, monitoring is not performed.

- **INIT_SCRIPT** Simics script to be run using `run-command-file`. For example, use this to attach real networks to avoid doing so after enabling reverse execution (which should be avoided due to Simics foibles).
- **AFL_PAD** Cause data received from AFL to be padded to this minimum packet size, intended for services having a minimum udp packet size. This value will also be used to determine the size of writes to receive buffers when multi-packet AFL is used.
- **AFL_UDP_HEADER** Used with multi-packet AFL, will split data received from AFL at these strings. If single-packet UDP sessions are desired, leave this undefined. NOTE: initial data, e.g., AFL seeds should include the UDP header, (vice relying on filters) because the system will split input data into datagrams at the headers *prior* to applying filters. The default maximum number of UDP packets is 10. A maximum is needed, otherwise AFL gets carried away by duplicating the header. (TBD, add an env override.)
- **AFL_PACKET_FILTER** A packet filter to modify or weed out data received from AFL for purposes of constraining fuzzing sessions to particular paths, e.g., to fuzz a single command dictated by a byte at a fixed offset. The value should be a .py file name, less the .py extension. The file must be relative to the Simics workspace and it must have a `filter(data, packet_num)` method that returns the desired string. For example, if the packet is to be rejected, return a byte array of zeros. Filters can also be used to compute and store CRCs within data.
- **AFL_OUTPUT** – Optional path to an AFL afl-output directory that will be referenced when running `playAFL` and `crashReport` commands. Typically this should not be used, and the system will use the value of **AFL_DATA** defined in the `.bashrc`.
- **AFL_MAX_LEN** – Maximum size of a data injection for AFL, into either kernel or user buffers. Note the trimming occurs just prior to injection, and thus queue files and records of recent data, e.g., the `ihung` file, will not yet be trimmed.
- **AFL_MAX_PACKETS** – Maximum size of a AFL packets written to the user space buffer.
- **AFL_STOP_ON_READ** – Stop fuzzing or tracking if the a read system call is encountered for the target FD after the fuzzed data is consumed.
- **AFL_STOP_ON_CLOSE** – Stop fuzzing or tracking if the FD is closed. AFL queue files will be given a suffix of *closed*.
- **AFL_SKIP_READ_N** – Experimental... Cause every Nth read/recv to cause RESim to skip over that kernel call. The intended use is odd cases where a TCP session does a series of reads, then a read with a large length, but then goes back and reads a deterministic amount of data.
- **TARGET_IP** – IP address to use on driver fuctions such as `replayAFL`
- **TARGET_PORT** – Port to use on driver fuctions such as `replayAFL`
- **STOP_ON_READ** – Cause AFL and replays to stop once the original read/recv call (or select) is hit.
- **READ_LOOP** – What to do if dataWatch detects a read loop, e.g., a vary large CRC calculation. The read loop trigger defaults at 10,000 and is overridden by the **READ_LOOP_MAX**. A value of *quit* will cause the RESim session to exit (useful for `crashReport`). Otherwise, the data watch is stopped and execution continues, e.g., to arrive at a page boundary fault.
- **READ_LOOP_MAX** – Limit on dataWatch reads from a single buffer.
- **EXECUTION_JUMPERS** – Name of a file with jumpers (see the `jumper` command in 3.7), expressed as from/to addresses in hex. **Note:** encountering a jumper will reset the reverse execution origin, and thus may not be suitable when using `injectIO` or crash analysis.
- **RESIM_LOG_LEVEL** – Set to INFO to supress debug messages in the resim log.

4.2 Target sections

Each computer within the simulation has its own section. The section items listed below that have a `$` prefix represent Simics CLI variables used within Simics scripts. If you define your own simics scripts (instead of using the generic scripts included with RESim), you may add arbitrary CLI variables to this section.

- **\$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`
- `$disk3_image` Path to the 3rd disk
- `$disk3_size` Size of 3rd `disk_image`
- `$mac_address_0` Enclose in double quotes
- `$mac_address_1` Enclose in double quotes
- `$mac_address_2` Enclose in double quotes
- `$mac_address_3` Enclose in double quotes
- `$eth_device` Alternet ethernet device, see [4.4](#) below.
- `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.
- `OS_TYPE` Either `LINUX` or `LINUX64` RESim session.
- `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.
- `BOOT_CHUNKS` The number of cycles to execute during boot between checks to determine if the component has booted enough to track its current task pointer. The intent is to keep this value low enough catch the system shortly after creation of the initial process. The default value is 900,000,000, which is too large for some ARM implementations. While components are booting, RESim uses the smallest `BOOT_CHUNKS` value assigned to any component that has not yet completed its boot.
- `DMOD` Optional file to pass to the `runToDmod` command once the component has booted. See [5.5](#).
- `PLATFORM` One of the following: `x86`; `arm`; or `arm5`
- `INTERACT_SCRIPT` Simics command script to be run after this component is loaded.

4.3 Target platforms

Targets platforms from the RESim `simics/simicsScripts/targets` include:

- `x86-x58-ich10` – An x86 platform with legacy BIOS that will boot many different x86 bootable disk images. Examples can be found in [section 8](#).
- `arm` – The Simics ARM QSP platform. By default this boots the Simics-provided QSP kernel. Alternately you can set the `kernel_image` parameter to point to a different kernel. See the directory at: `simics/image_build/arm-qsp` for information on rebuilding the ARM QSP kernel. The `root.disk_image` should point to a root file system. A stock root file system can be found at: <https://nps.box.com/v/resim-arm-qsp-rootfs> The user_disk—image appears as `/dev/qspdc` when the stock QSP kernel boots. If it contains a partition table and file system, then the first partition is mounted on `/mnt` by the stock root fs as `/dev/qspdc1`.
- `integrator-cp` – A 32-bit ARM platform modeled after the Integrator-cp. A kernel and sample `initrd` image are at:
<https://nps.box.com/v/resim-arm5-images>

4.4 Network definitions

Configuring the networks for a simulated system can require some trial and error and use of Wireshark to determine which logical ethernet devices are communicating on which simulated devices. If you are using the predefined driver component, see the Appendix P for mappings of addresses and device names. You may also start Wireshark from the Simics command line to observe which interfaces are connected to the different switches, e.g., `wireshark switch0` will display traffic on switch0. Take care to note MAC addresses and don't be fooled by packets routed through computers to other switches.

Four network switches are created, named switch0, switch1, switch2 and switch3. And two network hubs, named hub0 and hub1, are created. Each generic RESim computer has one or more network interfaces, depending on the type of platform. These are named eth0, eth1 and eth2, and are assigned corresponding MAC addresses from the RESim configuration file. By default, each computer ethernet interface is connected to its correspondingly numbered switch. This topology may be modified via entries in computer sections of the RESim configuration file. For example, an entry of:

```
ETH0_SWITCH=switch2
```

would connect the eth0 device to switch2. A switch value of `NONE` prevents the ethernet device from being connected to any switch. Note there is no error checking or sanity testing. In the above example, you would also need to re-assign the eth2 device or it will attempt to attach two connections to the same switch port. Alternately you can connect the ethernet devices to the hub, e.g.,:

```
ETH0_SWITCH=hub0
```

Ethernet devices on the generic x86 platform default to the `i82543gc` device defined by Simics. Use of the `$eth_dev=` entry lets you pick one of the following alternate ethernet devices:

```
i82559
i82546bg
i82543gc
```

If using a VLAN, select the 82559. The `i82543gc` does not properly support VLANs. Note that device changes ethernet names, e.g., `ens11` becomes `ens11f0` and `ens12` becomes `ens12f0`.

Simics CLI variables are assigned to each computer ethernet link using the convention `$TARGET_eth0` where `TARGET` is the value of the configuration file section header for that component. Similarly, connections from the computer to the switches are named using the convention `$TARGET_switch0`. These CLI variable names may be referenced in user-supplied scripts, or in Dmod directives of type `match_cmd`. See 5.5.

The `eth1` cli name is assigned to the motherboard ethernet slot. the eth0, eth2 get northbridge slots and eth3 gets a southbridge pci slot.

The `showLinks` command displays symbolic names of network connections. These can be used in DMOD directives, or manually to alter the network topology using the Simics `disconnect` command, e.g.,

```
disconnect $box1_eth0 $box1_switch0
```

4.4.1 Configuring VLANs

Create a VLAN switch by adding a `VLAN_N` to the ENV section of the ini file, where N identifies the switch. For example:

```
VLAN_1=3
```

will create a VLAN switch named `vswitch1`, and it will create a VLAN group id of 3 on the switch. Subsequent entries naming the same switch can be used to add additional VLAN groups to the switch. Then, when linking ethernet devices to the switch in the platform section, append the desired group number to the switch name. For example, the following will create a trunk connector for VLAN group 3 on switch `vswitch1`:

```
ETH0_SWITCH=vswitch1-3
```

To create a non-trunk connection (i.e., one that will not have VLAN group indicators in the ethernet frames), preface the group number with `nv`, e.g.,

```
ETH0_SWITCH=vswitch1-nv3
```

4.5 Driver component

Each simulation can have an optional driver component – designated by assigning 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** to be 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. An example driver is described in Appendix P.

5 Running the simulation

RESim sessions are started from the Simics workspace using the **resim** command (the path to which should be in your PATH environment variable).

RESim is controlled either through the simics command line, or using IDA Pro as described below. Interaction with the simulated computers, e.g., ssh'ing to a driver computer, requires that the simulation be running and not paused.

5.1 Installation

RESim assumes you have installed and are somewhat familiar with Simics. Versions 4.8, 5 and 6 are supported. It also assumes you have IDA Pro. See the *RESim Remote Access Guide* for information on remote access to RESim servers. RESim can be used without IDA Pro, though you are limited to the command line and don't benefit from function analysis provided by IDA.

Note if you are using an NPS-provided RESim server, this RESim installation will have already been done. But see the IDA Pro installation below for information on installing IDA.

- Get RESim from the git repo:

```
git clone https://github.com/mfthomps/RESim.git
```

- Install python-magic from gz file: pip install <path>

```
sudo pip install /mnt/re_images/python_pkgs/python-magic-0.4.15.tar.gz
```

- Install xterm

```
apt-get install xterm
```

- Define environment variables. In your **.bashrc**, define the following (place prior to the non-interactive return so they are picked up by ssh):
 - RESIM_DIR Path to your git clone.
 - SIMDIR Path to the Simics directory, e.g, **.../simics6/install/simics-6/simics-6.0.89**
 - IDA_DIR Path to your IDA installation directory (define on system where you will run IDA.
 - RESIM_IDA_DATA Path to directory to hold data use by RESim and IDA. Typically **\$HOME/ida_data**
 - AFL_DIR Path to directory containing the AFL executable (only needed to support fuzzing.)
 - AFL_DATA Path to directory containing the AFL seeds and output data.
- Add **\$RESIM_DIR/simics/bin** to your path, e.g. by adding to your **.profile**,

```
export RESIM_DIR=[path to your git clone]
export PATH=$RESIM_DIR/simics/bin:$PATH
```

5.1.1 Missing Simics platform files

The following files may be missing from some versions of Simics 6:

```
(install directory)/simics-6/simics-qsp-arm-6.0.*/targets/qsp-arm/images/qsp.dtb
(install directory)/simics-6/simics-qsp-arm-6.0.*/targets/qsp-arm/images/uImage
```

Contact Wind River to get those files (or contact mfthomps at nps.edu).

5.1.2 IDA Pro installation

The following steps augment a standard IDA installation (which is not covered here.)

- The IDA Pro installation should be configured to use Python 2.7, which must be installed, e.g., using `apt-get install python2-dev`.
- If IDA and RESim run on different computers, e.g., remote access to RESim:
 - Clone the RESim git repo onto your local computer (i.e., where IDA will run)
 - Configure the `RESIM_DIR`, `RESIM_IDA_DIR` and `IDA_DIR` variables as described above.
 - Extend your `PATH` to include the `RESIM_DIR/simics/bin` as described above.
- In your `$IDA_DIR/cfg` directory, there are a number of xml files that need to be replaced with those found in `$RESIM_DIR/simics/ida/cfg`. Backup the original xmls first.
- Start Ida and configure the debugger to use remote gdb, and configure the Debugger process options to use localhost and port 9123.

5.2 Getting started

Steps to define and run an example 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 using the `resim_ws.sh` script:

```
mkdir mywork; cd mywork
resim_ws.sh
```

2. Copy any desired files from `$RESIM/simics/workspace` or other workspaces into the new workspace. You may also create links to checkpoint directories in other workspaces.
3. 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 (unless the param file already exists, in which case you are done with initial setup).
4. Launch RESim using `resim mytarget`. That will start Simics and give you the Simics command prompt.
5. Continue the simulation until the kernel appears to have booted, then stop. Try stopping before the initial processes complete, i.e., while output is still spewing on the boot console.
6. Use the `@gkp.go()` command to generate the parameter file. This may take a while, and may require nominal interaction with the target system via its console, e.g., to schedule a new process. If it displays a message saying it is not in the kernel, try running ahead a bit, e.g., `r 10000` and try the `gkp.go` command again.
7. After the parameters are created, quit Simics and remove the `CREATE_RESIM_PARAMS` parameter.
8. Restart using the `resim` command. 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 [3](#).

5.2.1 Kernels with ASLR

The kernel parameter management mostly accounts for ASLR by tracking differences between addresses recorded when the parameters were first captured, and addresses during the current boot, e.g, using the FS base address. However, if the disk image containing the kernel is modified, the addresses may change in a manner that is not accounted for. Repeat the kernel parameter creation steps to mitigate that.

5.2.2 Kernel Parameters for 32-bit compatibility

If a 64-bit Linux environment includes 32-bit applications, first create kernel parameters per the above, and then run until one of the 32-bit applications is scheduled and use `@cgc.writeConfig` to save the state. Modify the ini file to restore that state and set `CREATE_RESIM_PARAMS` to YES. Then start the monitor and use `@gkp.compat32()`. This will modify the kernel parameters in the `.param` file to include those needed to monitor 32-bit applications.

5.3 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.

Start ida using the `runIda.sh` command (the path to which should be part of your `PATH` environment variable). If IDA will run on a different computer than where Simics is running, include the host name as a command line option.

The first time you start IDA, use the **Debugger / Process options** to ensure your host is localhost and the port is 9123. Save those as the default. Then go the **Debugger setup** and select **Edit exceptions**. Change the **SIGTRAP** entry to pass the signal to the application; and to only log the exception. Save the settings. You should only need to do this step once.

The first time you open a given program in IDA, run this script (from File / Script file):

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

This will create a data file used by RESim when generating stack traces. You may also run the `findBlocks.py` script to generate a database of basic blocks that will be used by RESim for tracking code coverage and to instrument RESim for use with AFL.

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 `exec`'ed and execution is transferred to the text segment, at which point it will pause. Assuming you started IDA with the `runIDA.sh` command, you may now press `shift-r`, which will cause IDA to attached to the process and load its RESim plugin. Alternately you can manually attach the process and run the IDAPython script at:

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

Note: When you use `shift-r` or attach the debugger, IDA may generate a few pop-ups informing you of this or that. When that happens, select the *do not show again* checkbox. Then, save the IDA state, kill it and restart it. The goal is to suppress those pop-ups, which can corrupt the loading of IDA plugins when the debugger is attaching.

You can now run the commands found in the debugger help menu. Those commands generally invoke RESim commands listed in 3.

The RESim IDA client is not a robust debug environment in the sense that you can easily cause Simics to leave your intended execution context. There are attempts to catch the termination of the process being debugged. But in general, you should consider defining bookmarks to allow you to return to a known state.

There are situations where it is most productive, or necessary, to engage with the Simics command line directly. If you change the execution state via the command line, you can get IDA back in synch via the **Debugger / Resynch with server** menu selection. Or you can double-click a bookmark or watch mark.

RESim commands are available in IDA via the Debugger menu item, and via right clicking on addresses.

Setting IDA breakpoints requires that the current context be within the thread family being debugged. For example, if execution is currently stopped in some other process, use *Run to user space* to get execution back into the debugged process. You may then use IDA breakpoints.

The IDA step-over function will usually handle page faults and return you to the point after which the page is swapped in. If you find yourself in the kernel, use *Run to user space*.

5.3.1 RESim IDA windows

The IDA plugin creates several tabs along side of the IDA *Stack view* window.

- Bookmarks – List of bookmarks created by the user (e.g., via right click), or created automatically when reverse tracking data. Double-clicking on a bookmark will skip execution to the associated cycle.
- stack trace – The stack trace, as determined by RESim. You may need to right-click / refresh to see this. Double-clicking on a item will take the *IDA View-EIP* to that address, but has no effect on the current execution cycle. Note, your previous focus must be in the *IDA View-EIP* window, otherwise you may end up changing the address of a *Hex View* window.

- data watch – List of watch marks generated by a trackIO or injectIO (which would be run from the Simics command line). Use right-click / refresh to view these. Double-click will skip the execution to the associated cycle.

5.3.2 RESim IDA options

Provide the `color` option to the `runIDA` command to cause IDA's graph display to be colored based on the content of the programs hits file found in the `$IDA.DIR`. Note that after running the `runPlay` or similar functions, you must copy the newly created `program.target.hits` file to the `program.hits` file.

The last option provided to the `runIDA` command can name a server on which RESim is running. The script will create an ssh tunnel to that server using port 9123. Note that some programs require a lot of data exchange between the gdb server and the client, and using an ssh tunnel can result in poor response times. Consider using the IDA Debugger/Process options to set the the host explicitl.

5.4 Ghidra

A preliminaray RESim plugin for the Ghidra debugger is available at <https://github.com/mfthomps/RESimGhidraPlugins>. See that repo's README for installation instructions. It is assumed that the user has experience with the Ghidra platform.

Use of the Ghidra plugin requires a modified version of gdb, available at <https://github.com/mfthomps/binutils-gdb>. The modification causes gdb to display responses from gdb "monitor" commands using the same FD as used for other gdb command results. This is needed for Ghidra to see the results of monitor commands, which the plugin uses to interact with RESim.

Once the RESimGhidraPlugins extension is installed, start the debugger and organize the RESim windows as desired, e.g., by dragging them to other tabbed windows.

Use the RESim / Configure menu options to set the path to your customized gdb; the path to the file system root of the target binary, the host:port of your Simics host, and to set the ARM architecture if needed.

The windows described for the IDA plugin are duplicated within the Ghidra plugin.

Navigation and functions are available in the RESim menu, and by right clicking. For example, to reverse track the source of a memory location, put the cursor over the operand and right click...

Ghidra provdes logs, typically at `$HOME/.ghidra/.ghidra_10.3_DEV/application.log`

Ghidra stores its analysis in "project" directories, typically in home.

TBD: Automatic copy of RESIM_IDA_DATA from RESim host to Ghidra host. If host configured as localhost, look at the tunnel (or do copy as part of creating tunnel?).

5.5 Dynamic modifications to memory and topology

RESim includes functions that dynamically modify modeled elements and connections, triggered by system events. For example, a script that loads selected kernel modules could be augmented in memory to load alternate modules, e.g., those for which you have modeled devices. Modifying such a script on the volume image itself is not always convenient, e.g., *tripwire* functions might manage checksums of configuration files. It is therefore sometimes preferable to dynamically augment the software's perception of what is read.

The `runToDmod` function triggers on the reading or writing of a specified regular expression via the `write` or `read` system calls. The `runToDmod` function includes a parameter that names a file containing Dmod directives. In all subfunctions listed below, the `match` string identifies the read or write operation that triggers the action. The format of directive files depend on the subfunction. Each subfunction also identifies whether it is triggered on a `read` or `write` operation.

- `sub_replace <operation>`– Replace a substring within a read or write buffer (specified by the `<operation>`, with a given string. The directives file includes one or more sets of directives. The directives use regular expression syntax. An example directives file looks like:

```
sub_replace read
#
# 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.

- **script_replace** <operation>- Replace a substring within a script buffer with a given string. The intended use is to dynamically modify commands read from script files. Some implementations read 8k from the script file, operate on the next no-comment line, and then advance the file pointer and repeat. This causes your Dmod target to be read many times. With a script_replace Dmod, the target match is only considered when it matches the start of the first non-comment line of a read buffer. The directives file includes one or more sets of directives. The directives use regular expression syntax. An example directives file looks like:

```
script_replace read
#
# match
# was
# becomes
modprobe e1000e
modprobe e1000e
modprobe e100
```

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

- **full_replace** <operation> - Replace the entire write or read buffer with a given string. The directives file includes a single directive whose replacement string may include multiple lines.

```
full_replace write
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** <operation> Execute a list of Simics commands when the trigger string is found in a read or write buffer (per the <operation> field), 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. Simics commands may reference CLI variables defined via the RESim configuration files, such as network connection names described in 4.4.

```
match_cmd write
#
# match (regx)
# was (regx)
# cmd
KERNEL=="eth\\*", NAME="eth
("00:e0:27:0f:ca:a8".*eth1|"00:e0:27:0f:ca:a9".*eth0)
disconnect $TG1_eth0 $TG1_switch0
disconnect $TG1_eth1 $TG1_switch1
connect $TG1_eth0 cnt1 = (switch1.get-free-connector)
connect $TG1_eth1 cnt1 = (switch0.get-free-connector)
```

- **open_replace** open If a given file or device is opened and fails (because it is missing), the return from open masks the failure and returns FD 99. When the application reads from that FD, the content of a data file is returned to the application.

```
open_replace open
#
# fname
# len
# becomes
/dev/mtd1
0x0
uboot_params.data
```

The `becomes` field is the name of a data file whose content is returned to the application when reading from the FD associated with the missing file. If the `len` field is 0, the length of the file named by `becomes` is used.

5.5.1 Dynamic modifications to multiple computers

Dynamic modifications (Dmods) to system state are often performed early in the boot process, e.g., as network devices are being assigned addresses or kernel modules are being loaded. When simulated components boot, RESim monitors them to determine when each has booted far enough for the current task record to be of use, which is typically when the `init` process runs. RESim then pauses after all computers in the simulation have reached this initial state. Note though that the first computers to have reached their initial state will continue on while other computers are still booting. Thus, by the time RESim pauses, some computers may have executed beyond the point at which a dynamic modification was desired.

To avoid such race conditions, the RESim configuration file can optionally include the `DMOD` directive to identify Dmod files for each of the computers in a simulation. If present, the `runToDmod` command is executed as soon as the corresponding computer reaches its initial state. This ensures that dynamic modifications to those computers will occur while other computers in the simulation continue to boot to their initial states.

5.5.2 Dmods in the background

When dynamically analyzing a process or family of threads, RESim generally manages breakpoints for just those processes. This optimization can significantly speed up the analysis processes by entirely ignoring system calls and other events that occur in processes other than those under analysis. However, there are times when you need to dynamically modify some other process while debugging. Consider this example: you've directed RESim to debug some program X, and RESim has now detected the loading of X and has broken execution. You now want to observe X, e.g., tracking its I/O to some socket, however you know that at some point in the coming system execution, some other process Y will write a value that X will observe (perhaps indirectly). Use of the `background=True` option with the `runToDmod` command will cause RESim to monitor all system calls associated with the Dmod, even while you focus on the behavior of X.

5.5.3 Manual modifications to topology

Simics `connect` and `disconnect` commands may be used at any point in the simulation to modify the network topology. See the discussion in 4.4 for naming network connections. Potential uses of this include:

- Remove real network connections that had been used to provision the simulation, i.e., to avoid real-world leakage into the simulation and resulting corruption of reverse execution.
- Disconnect computers that frequently send data to network ports of interest, i.e., to reduce noise and simplify your ability to send your own data to those ports.

5.6 Sending data

There are a variety of ways to send data to a target. The quick and easy way is to use the Simics real networks, which utilizes port forwarding. For example, if the target network is connected to `switch0` and the target port is 62226 and the IP is 172.16.8.100, the following will forward traffic from the localhost port 62226 to the target:

```
connect-real-network-port-in target-port = 62226 host-port = 62226 \  
ethernet-link = switch0 target-ip = 172.16.8.200
```

The use of real networks has limitations described in S.1.

It is often preferable to send data from a driver component as described in P. If you use the default RESim driver, then the `drive-driver.py` script provides a convenient way to send data from a driver to the target.

5.7 Tracking data

RESim provides several functions for tracking data consumed by processes. A `Data Watch` data structure tracks input buffers into which data is read, e.g., automatically as the result of a `runToIO` command. The `watchData` command causes the simulation to proceed until any of the `Data Watch` structures are read – and also lets the analyst create new Data Watches. The `showDataWatch` command displays the current list in terms of starting address and length. The `trackIO` command automates iterations of `watchData` commands, creating a list of execution points at which `Data Watch` structures are read. The resulting `watch marks` are bookmarks that can be view using `showWatchMarks` and skipped to using the `goToDataMark` command, naming the index. The IDA

client displays these in its **data watch** window. These data watch bookmarks are distinct from the bookmarks viewed with the **listBookmarks** command, and displayed in the IDA **Bookmarks** window.

The **trackIO** function also dynamically creates new **Data Watch** structures as input buffers are copied into other buffers. It recognizes (some) common data copy functions, (e.g., **memcpy** and **strcpy**).

The **trackIO** uses a *backstop* value to determine when to stop looking for additional input or references to Data Watch buffers. The value is in cycles, and thus useful values can vary wildly between environments. Use the **BACK_STOP_CYCLES** value in the RESim configuration file to adjust this. To track multiple packets, simply send them one after another before the backstop is hit. The backstop is not employed until the first read on the given FD is encountered (so no reason to hurry your network traffic). The backstop is cleared whenever it is hit.

Note that in many situations, by the time the backstop is hit, the thread has invoked a syscall. If it is a blocking **recv**, then the next **trackIO** command query a record of the previous system call (maintained by **reverseToCall**) so that it can properly record and track the call.

When a data watch buffer within the stack appears to be substantially overwritten, it is automatically removed from the watched buffers. The associated heuristic is intended to catch reuse of stack buffers to avoid false watch marks. Beware of false removals, i.e., you may miss accesses to buffers.

5.7.1 Watch Marks

Watch marks identify references to input data. Each mark describes the reference. For example:

```
18 Read 4 from 0x0837eb3c offset 12 into 0x 837eb30 (buf size 20) ip:0x8072c22
```

describes a reference that occurs at IP 0x8072c22. It says that four bytes were read from 0x0837eb3c, which is twelve bytes into some input buffer (or copy thereof) that begins at address 837eb30 and is twenty bytes long.

Copying of buffers is noted by use of *Copy* instead of *Read*. A copy is typically due to a **memcpy** or **strcpy**. If the phrase *Ad-hoc* occurs within the watch mark, that implies the copy occurred via a sequence of moves from memory to a register and then to a different memory location. A *Write* watch mark indicates the content of a watched buffer is modified. Watch marks may also reflect string comparisons, e.g., via the **strcmp** function.

There may be cases where RESim reports references to buffers that occur after the buffer was reused, e.g., a stack buffer. RESim catches some, but not all, cases of stack buffer reuse.

When injecting data, e.g., via **injectIO** (see the following subsection), the list of referable watch marks is reset on each data injection. Thus, if the **injectIO** function injects three packets, only the watch marks recorded after the third packet will be in the list of watch marks that can be skipped to. The older, stale watch marks can be viewed using the **old** flag in the **showWatchMarks** command.

5.7.2 Tracking injected data

Once a **trackIO** is performed and you've reviewed input data references, you can repeat the tracking step with modified data:

- Skip the simulation to the point at which the original data was read, e.g., following the **recv** syscall. So this by double clicking on that Watch Mark, or use the **goToDataMark** function in RESim.
- Use **modifyMemory** to change data in memory. Note this will reset the reverse execution bookmarks, preventing you from skipping to any earlier point.
- Use the **retrack** function.
- You may repeat this as many times as needed, however instead of skipping to a Watch Mark, you can simply skip to the **origin** bookmark since that is now the point at which you previously modified memory.

A related function, **injectIO**, injects data from a file into a read buffer identified via the **prepInject** function, which creates a snapshot and stores information related to the data input, including the original buffer data. RESim includes a suite of related data tracking functions to replay AFL queue files.

5.8 Preparing for data injection

Data injection techniques depend on whether the goal is to inject data into the application buffer, or into a kernel buffer.

The **prepInject** command is used to put the simulation in a state suitable for injecting data into an application read buffer, i.e., the named by the application when making a syscall such as **recv**. For network traffic, the general steps for this include:

- Run the simulation to a desired state after selecting the target process for debugging, e.g., using `debugProc`. This state may include completion of initialization that may occur while the target process is waiting in the kernel on the target `recv`.
- Create a snapshot.
- Prepare a means of sending sample data to the target port. The purpose of this data is to cause the target to receive it and return. The data itself will be replaced later by injected data. The data should not include multiple UDP packets, you want a single read.
- Restart RESim and issue the `prepInject` command, providing the FD of the receive socket a name for the new snapshot.
- Send the sample data.
- RESim will pause the simulation when the data is read and will create a new snapshot.

In general, data should be sent to the target from a driver or other simulated computer rather than via a real network. Use of real networks to send data for `prepInject` may fail because of the problems associated with real-world leakage, see [S.1](#). Even using a real network to `ssh` into a driver to send the data may lead to corruption, particularly if the target process is waiting in the kernel for data. The use of `ssh` and `scp` can be made safe by executing the `simics/magic/simics-magic` binary on an x86 driver just after starting `prepInject` but prior to sending data to the target. That program executes a Simics *magic instruction* that RESim detects and uses to reset the origin for reverse execution during performance of `prepInject`. The use of this magic is currently required for all cases where the receive call is waiting in the kernel, whether or not `ssh` is used. See the `simics/bin/driver-driver.py` program for an example of using the magic instruction when sending data from the driver. Using the `-d` option will cause it to establish a new origin and will disconnect the driver from the network simulation.

5.8.1 Tracking data through pipes

The `postscripts/pipe.py` script creates a `pipes.txt` file (see [3.2](#). RESim looks for this file in the workspace and will track data through the pipes defined therein. For example, if a thread reads data from a network and then writes that data to a pipe, RESim will then watch for a thread to read data from that same pipe.

5.8.2 Tracking libraries

The data references of interest might occur in a shared library. See section [C](#) for information on opening IDA to observe library references to data.

5.8.3 When are you done?

When tracking data, you need to decide when to stop looking for data references. The primary mechanism for this is the `BACK_STOP_CYCLES` setting. When tracking data, RESim declares it is done tracking after that number of execution cycles occur with no data references. Note this can mask interesting behavior (and faults) if the number of cycles is too small. Note also that failure to account for all data buffers can lead to tracking ending prematurely. When in doubt, test with ridiculously large cycle counts.

5.8.4 Warning

Use of real networks to generate tracked data can lead to silent corruption of the simulation unless care is taken to reset the reversing origin after the real network is done interacting with the system. See [S.1](#).

Also, modifying memory content or register content will effectively redefine the origin used for reversing. Do not use direct Simics commands to modify memory or registers, use the RESim wrappers.

5.8.5 Tracking backwards

RESim can track the origins of data named by a register or memory address using the `revTaintReg` and `revTaintAddr` functions, which are also available via the IDA client. These functions do not generate data watch bookmarks, rather, they generate bookmarks viewable with `listBookmarks` or in the IDA Bookmarks window. RESim will backtrace the data until one of three events:

- It finds a kernel read or `recv` call that wrote the data into memory;
- The origin is reached, i.e., the start of reverse execution; or

- An operation on the data is not easily traced to a previous memory location. Examples are multiplication and boolean transformations.

If the origin is reached, RESim will query the Watch Marks to attempt to trace the data back through buffer copies. If successful, it will identify the initial read buffer (e.g, the 3rd read) and the offset into that buffer. This output will appear at the Simics command line, or in the Ida output window, depending on where the command originated.

5.8.6 Removing unwanted traffic

Target network ports may receive periodic traffic from other components in the simulated topology, thus interfering with your crafted data streams. Offending network connections can be disconnected as described in 5.5.3.

5.9 Code coverage

RESim generate two different kinds of code coverage artifacts:

- Coverage files resulting from the replay of AFL sessions using the `runPlay` script. These files are stored beneath the AFL output directory and include cpu cycles and UDP packet number for each hit.
- Hits files containing lists of all basic blocks hit in a session or cumulatively.

5.9.1 Coverage from fuzzing

After a fuzzing session, the `playAFL` script will create coverage files for each code execution path found by AFL. These tend to have a lot of redundancy because of the way that AFL determines uniqueness. When fuzzing UDP with a defined `AFL_UDP_HEADER`, the extraneous execution paths compound, e.g., if AFL gets 2 UDP packets for one session, and each of result in hit sets already covered by different other sessions, AFL will treat that as a execution new path. For this reason, the quantity of UDP headers is bounded as described in TBD.

The `dedupCoverage.py` script creates a list of coverage files that are, in some sense, unique. First, removes duplicate coverage files resulting from parallel fuzzing. Next it removes any file whose hits are a subset of another coverage file. When UDP headers are used, the script first assesses all files from sessions having a single UDP packet. It then considers multi-packet sessions that result in new hits. And where these match or are subsets, it removes files giving preference to those whose hits occur on the earliest session.

5.9.2 Ad-hoc code coverage

Features to reflect code coverage should support human analysis as well as automated analysis, e.g., fuzzing. To support the human, coverage is highlighted within IDA using color-coded basic blocks. The scope of coverage is tied either to the text segment, i.e., the program, or a single shared library. We refer to this as a *coverage unit*. Coverage tracking commences with use of the `mapCoverage` command and is intended to aid understanding of program response to inputs tracked via the `trackIO` or `injectIO` commands. Those basic blocks which are only hit between coverage commencement and the first input event are separately colored so as to not be confused with blocks hit subsequent to receipt of input. This distinction is intended to facilitate comparisons between *data sessions* defined as periods between receipt of input (or IO injection) and quiescence with respect to the backstop logic. Within the IDA display, coverage properties of basic blocks will be coded into five colors:

- Never executed.
- Never executed during a data session
- Executed during a data session, but not during this data session
- Executed during this session and some previous data session.
- Executed only during this data session.

The `coverage` module maintains three data files for each coverage unit: a running total set of hits from previous data sessions; the hits for the most recent data session; and hits which occur prior to the first input data reference (e.g., reads, selects, etc). The running total is only updated immediately prior to the start of a new data session. At the start of each data session, the coverage module will restore any breakpoints found in the recent session data file and then merge it into the running total. The coverage module will then clear its internal hits data. When the IDA client is started with the `color` argument, it will first reset all basic block colors, and will then read the hits files and color blocks accordingly. The running hits file is intended to persist across RESim sessions.

5.9.3 Branches not taken

The IDA client generates a list of basic block branches that were not taken, i.e., unused exits from hit basic blocks. This list is presented in the BNT window of the IDA client and is intended to help the analyst see places where changes in input data may have resulted in added coverage. When used following a RESim `mapCoverage` command, double-clicking on an entry will take the simulation to the corresponding cycle. Note this is only available for the very first coverage hit of each basic block, and of course only within a session that hit the from block (see the `bbAFL` function). This window is most useful when IDA is started with the `color` option.

The `color` option and BNT window can also be used outside of the `mapCoverage` context, e.g., to analyze branches not taken during AFL sessions. Double-clicking an entry in the BNT window will go to the source of the branch not taken (note the simulation state is not altered.) The following is a notional code coverage workflow.

- Use AFL to fuzz a target.
- Use the `playAFL` command to replay all of the sessions, generating a new hits file.
- Copy that hits file over the target hits file.
- Start a RESim session from the checkpoint used during AFL.
- Start IDA and use the `resetBlocks` script to clear block coloring; then use `Color blocks` to color all blocks hit during the AFL session.
- Find an interesting BNT in the BNT window.
- Use the `findBB.py` utility to search for all AFL data files that hit the source block.
- Restart RESim and use the `injectIO`, `aflInject`, `aflTrack` or `aflReplay` functions to play the data file identified above.
- Run to the source block of interest and investigate whether an altered input could result in a branch to the BNT.

Note in the above workflow, you may wish to use 2 different checkpoints. The `playAFL` function must run from the checkpoint used with AFL. However the other functions that use `trackIO` could run from checkpoints that include a driver component. And that checkpoint might be from early on in the process's life, e.g., to identify the sources of state values that do not seem to result from the input data.

Initially, no database of hits will be maintained tying data files to hit sets. A downside is that if a data file is consumed twice, the analyst no longer sees which basic block hits are unique to that data file. Each subsequent data session artifacts are additive, and the effects of any previous runs cannot independently viewed.

5.9.4 BNTs from fuzzing

The `findBNT.py` utility displays BNTs resulting from fuzzing sessions. It assumes you've played the AFL sessions, e.g. via the `runPlay` utility or `playAFL` at the Simics command prompt. It also assumes you've run `dedupeCoverage.py`. The resulting list of BNTs can be manually explored using the `injectToBB` RESim command. This will find a fuzzing input that reaches the given BB, play it and break at that BB. An example workflow for this might be:

- After fuzzing, use `runPlay` to replay and record the BB's from all the paths found by AFL. And run `dedupeCover.py` to generate a list of unique coverage files.
- Use the `findBNT.py` utility to list all of the BNTs and the BBs which lead to them.
- Run the RESim `injectToBB` command to take the simulation to the BB leading to the selected BNT.
- Start IDA and look at the branching conditions to identify data that would cause the program to take the BNT and guess if that appears to be a function of inputs.
- Use the IDA Client `reverse track data` function to find the source of the data. Note that if the input data includes multiple packets, RESim may still identify earlier packets as the source by tracing the data back through watch marks.
- Assuming you have an alternate input value to test, and an offset within the input file as reported by RESim, use `hexedit` to modify the input file copy created by `injectToBB` at `/tmp/bb.io`.

- Use `injectIO` to confirm that the modified input file reaches the target BNT. For example: `debugSnap; doBreak; injectIO`.
- Run the `addInput.py` utility to add your modified `/tmp/bb.io` to the set of input files discovered by AFL.
- After the BNTs have been explored, use `playAFL` and `dedupeCoverage.py` to update the coverage records.
- Use the `cycleSeeds.py` utility to add all of the input files corresponding to unique coverage to the target seed directory. This will include your manual additions.
- Delete the `afl/output/<target>` files and restart AFL to see if it can use the new paths you discovered to discover more paths.
- Use the `arch-tars.sh` utility (edit to name your archive path) to archive the AFL results and the workspace.

The `-d` option of `findBNT.py` will annotate its output with any data watch marks found in the BB leading to the BNT. This option uses watch mark data generated using the `runTrack` utility, which runs `injectIO` on all of the AFL sessions and records all of the watch marks. This option can be useful when there are a large number of BNTs and you want to prioritize those with a higher likelihood of being affected by inputs.

5.9.5 Hits in libraries

The instruction addresses in hits files are based on static program values, i.e., when recording hits in a shared library the runtime address values are reduced by the image base. The RESim IDA plugin provides a `GoTo` function to jump to a program address computed by adding the image base to the given value. This is invoked using `Ctrl-Shift-g`.

5.10 Selecting checkpoints

Use of checkpoints can significantly simplify and speed up analysis, e.g., by not having to wait for simulated systems to boot each time. The performance and ease of analysis can depend on the state of the simulated system when a checkpoint is made. Assume you wish to analyze a particular service. After using the `debugProc` command to select it and gather its information, consider running the simulation forward for a while until you see that other system initialization has completed. Creating a snapshot after initialization may speed up subsequent analysis by reducing the amount of extraneous execution.

Watching the log at `logs/monitors/resim.log` can let you know when initialization has largely completed. While you are waiting for initialization to complete, the target process may be waiting in a kernel call, e.g., an `accept` or a `read`. RESim retains that information in its checkpoint. For example, when running from such a checkpoint, a `trackIO` function on the bound FD will catch the return from `accept` and alter the tracked FD to the new FD returned by the kernel.

6 Fuzzing with AFL

RESim is integrated with a forked version of AFL from `github/mfthomps/AFL`. The AFL fuzzer was modified to provide its fuzzed input to RESim via a network socket. RESim uses that socket to send AFL results of the fuzzing session in terms of basic block edges hit. The `-R` switch tells AFL that this is a RESim session.

RESim either injects data received from AFL directly into application memory, or it injects the data into a kernel read buffer, depending on the setup steps used to configure the fuzzing session. UDP and single-read TCP applications will typically use application memory. TCP applications that might issue multiple kernel calls to receive data should be configured to inject data into kernel buffer. When injecting fuzzed data, you must account for buffer sizes. With user space injections, the length value given to a `recv` call can be a guide (but not always). Injecting data past the end of either the kernel buffer or an application buffer is a fine way to observe artificial SEGV's.

The setup steps create a snapshot and associated data that dictate the starting state and the location to which data will be written.

Parallel fuzzing is supported as described in section 6.13. But you must first prepare and test a single fuzzing instance as described below.

- Clone the AFL repo (<https://github.com/mfthomps/AFL.git>)⁵ and use `make` to build the executable.
- Create a RESim workspace for the fuzzing session. The name of the workspace will be used to name fuzzing artifacts, so make it meaningful.

⁵Older environments may need to update binutils.

- Confirm your `bashrc` file defines the AFL paths per section 5.1.
- Create a *target* directory beneath the AFL output directory having the name of your workspace.
- Create a **seeds** subdirectory beneath that.
- Use tracing or static analysis to identify the FD the service binds to and whether packet processing requires a minimum sized packet.
- Use the **prepInject** command to create a snapshot of the state where data from the FD has been read and user space is being returned to. Typically, start the system from scratch and use **debugProc** to debug the desired process. Then issue the **prepInject** command and send the service some data (or wait for some other component to do so.) The data does not matter – though make it large enough to map any page that might be referenced by the maximum read length, and do not send multiple UDP packets. Update your ini file to reflect the snapshot name. See section 6.3.1 if injection into kernel memory is needed.
- Assuming you’ve created a set of input files that follow different paths through the service of interest, reduce them to the minimal number of bytes needed to function – or skip this step if you think your input files are as small as they need to be. Use the **fuzz** command for each of your input files to create a minimal set of seeds for AFL. Prior to using the fuzz command, Each run of the fuzz command creates an output file in `/tmp/trimmer`. You can run the fuzz command multiple times from the same RESim session (it returns to the original IO state.)
- Copy those trimmer files into the AFL seed directory for this target.
- Set values in the ENV section of your ini file per section 4.1. These may include a UDP header, and the backstop cycles that determine how long an AFL session will continue to run after hitting the the most recent basic block.
- Use the **runAFL** command, naming the target (e.g., your workspace name).
- You should see AFL run some initial sessions for calibration and then begin its fuzzing.
- Output from the Simics command line is redirected to `/tmp/resim.log`, check that for errors if AFL hangs. Also look at `resim.db.log` in the workspace.

Each fuzzing session will run until some number of cycles have elapsed since the most recent basic block hit. This value is defined in the `BACK_STOP_CYCLES` environment variable within the ini file. A session will also end if there is not more data to inject into application memory upon hitting the read call address. (NOTE: the backstop value is currently shared with the value used with `trackIO`, and thus may need to change for fuzzing.)

The `HANG_CYCLES` value determine when a session is considered to be hung, e.g., the program is in an infinite loop. Hangs may reflect more than just DoS vulnerabilities, e.g., corruption of a return address might lead to a viable code location that leads to a loop.

The `AFL_STOP_ON_CLOSE` env value will end the fuzzing session if the FD is closed, and `AFL_STOP_ON_READ` will end it if a read system call is encountered on the target FD after all fuzzed data is consumed.

6.1 Seeds

Well chosen seeds can save a lot of fuzzing time. For example, if all packets must start with a specific string there is no sense in making AFL discover that string. On the other hand, AFL can quickly discover many protocol values that affect execution paths from simple data. This is particularly true when the data sizes are small.

There is no one right way to select seeds. Sometimes a sample of PCAPs provide suitable seeds. However there are other cases where AFL would be more efficient with a smaller degenerate data set, e.g., a few key fields and a small set of a constant character. An example of such a case might be a protocol in which delimited fields are relatively long within the PCAP, but could be very much shorter in the protocol. In such a case, AFL might struggle to determine the field delimiters (not that it explicitly does any such thing) from the longer PCAP fields.

It is often the case that smaller seeds yield quicker results. There are though plenty of exceptions to this. An understanding of the protocol is

6.2 Multiple UDP packets

AFL is organized as a file fuzzer, and it is up to the user to adapt it (or the target) for network environments. Our fork of AFL uses networking instead of files to provide data to the RESim environment. AFL generates fuzzed data. RESim injects this data directly into the target memory to at a point where the target has returned from a `recv` syscall. For single UDP packets, we simply truncate the AFL data to conform to expected UDP packet size (trusting that AFL will not focus future resources on fuzzing truncated data.) For multiple packets, RESim catches subsequent calls to `recv`, injects the new data, adjusts the size in the return register and skips execution forward to the return from the system call. Handling multiple packets currently requires that we assume the `recv` calls are made from a common user space address, and the returns from `recv` all end up at the same user space address. This seems a safe assumption for most applications.

Splitting multiple UDP packets for each coverage session currently takes one of two approaches. If the application is known to have a fixed UDP header, e.g., `RrUDP`, then we can split AFL data at those strings using the `AFL_UDP_HEADER` environment definition in the ini file. Seed packet combinations can be concatenated to accommodate whatever packet quantity is desired. RESim will reject any packet that lacks the prescribed header for subsequent packets, which should cause AFL to see reduced coverage for such data sets.

The more challenging case is where the application expects a minimum packet size. That complicates creation of minimal seeds. If the minimum size is 1400 bytes, a 2 packet sequence would require a seed of 2800 bytes, which is a lot for AFL to fuzz.

6.3 Fuzzing TCP

Use the `aflTCP` command (or preferably the `-t` switch on the `runAFL` command) to start a session to fuzz TCP. When setting up the snapshot, use a driver computer rather than real networks, e.g., with `netcat`. Otherwise undefined behavior will result, e.g. if the process tries to send to the socket. TCP sessions initialized using the `prepInject` command assume the application gets its data from a single read call to the kernel. Applications that depend on the kernel to provide buffered input, e.g., character-at-a-time reads, or applications that otherwise make multiple kernel calls to read data should have data injected into the kernel buffer. This is achieved by using `prepInjectWatch`, as described in section 6.3.1

Note that while kernel data injection may provide the appearance of generating data for multiple application calls to the kernel, in practice many programs treat TCP as an application level packet delivery system, and depend on packetizing side effects. Consider the case where a client sends a command to the server, waits for a reply and then sends another command. A developer might implement the server while assuming only a single command would be seen with each read.

6.3.1 Partial reads

Consider an application that does not read all data available from an interface, e.g., one that reads a character at a time from the kernel. Such an application may violate assumptions made when injecting data directly into application memory, e.g., the assumption that the application will read all necessary data in a single read. For example, a non-conformant application may end up hanging on a `select` call because the kernel does not know there is more data to inject into the application memory. Another problem with partial reads is that they would break the use of `injectIO` since each injection into application memory resets the origin used with reverse execution.

The `prepInjectWatch` operation can be used to prepare snapshots to inject data directly into the kernel buffer instead of into application memory. Currently this assumes the ether that the application uses `ioctl` to determine how much data is in the kernel buffer, or that the initial buffer from the `prepInjectWatch` is as large as any data to be injected. The latter case, e.g., an application that simply reads a character at a time until it sees some delimiter, will stop the simulation when the application has consumed all the data provided in the injection and again hits the read call.

6.3.2 TCP timing and packetizing

Sometimes programmers make assumptions about TCP reads, e.g., they assume a relationship between the quantity of bytes written in a single write to the quantity of bytes read in a single read. Thus, changes in things like MTU can affect the application protocol...

6.3.3 State of the target system

Stateful services such as FTP typically require some amount of interaction to get the process into the state at which you'd like to fuzz it. In the example of FTP, this may include logging in, e.g., as *anonymous*, and issuing the `PORT` command. Keeping with the FTP example, you would create a driver component and `ssh` to that as

part of the setup. If you wish to fuzz commands that use PORT, e.g., LIST, you would login to the ftp server and then stop the simulation. Knowing that a command such as LIST will first send the port command, you would use `prepInject` with `count=2` to pause the simulation after it has first read the PORT directive and then the LIST directive. The fuzzing injection would then be overwriting the LIST directive.

Using AFL to fuzz a sequence of FTP commands is not currently possible.

6.4 Packet filters

Sometimes you may wish to focus the fuzzing on a specific code path determined by data within the packet. This may be of particular interest in multi-packet UDP cases where you know a specific command will lead to interesting paths when the 2nd packet is consumed. Left alone however, AFL will find interest in code paths generated by the first packet having other commands. Currently, you can define a `AFL_PACKET_FILTER` as a python program to reject AFL data that does not contain desired content, e.g., a specific byte at a specific offset. The program must have a `filter(data, packet_number)` method that returns data to be injected. For example, return nulls if the packet is to be rejected. The intention is that will yield poor coverage, causing AFL not see new paths. Alternately, return an altered packet, e.g., with a computed CRC injected into the data. When creating filter modules, you may need to modify the python module search path to include the python distribution, e.g.,

```
sys.path.insert(0, '/usr/lib/python3/dist-packages')
```

6.5 Address jumpers

Execution paths can be dynamically altered, e.g., to avoid CRC computations using the `addJumper` command. This takes as inputs the basic block addresses of a from-block and a to-block. When code coverage hits the from-block, the IP is modified to jump to the to-block. The jumpers are managed in a file with a `.jumpers` suffix along with the IDA databases and block lists for applications.

Jumpers are also useful when you discover very slow fuzzing sessions caused by large quantities of iterations driven by an input value. A `Packet filter` as described above might mitigate such a problem, however that can artificially constrain data whose value may have effects beyond the iteration count. Review of the iterated code should identify whether use of a jumper might affect program execution.

These jumpers are only intended for use with code coverage and/or fuzzing are different than those used with the `jumper` command described in 3.7, which provides more general alterations to execution control flow.

6.6 Fuzz another library

Use the optional `fname` parameter to name a shared library file (relative to the `RESIM_ROOT_PREFIX`). You must have first opened that file with IDA and used the `findBlocks.py` script to save a database of its basic blocks.

6.7 Fuzz another process

Consider the case where one process receives data and sends a derivative of that data to another process via a pipe or internal network, and you want the fuzzing guidance (i.e., code coverage feedback) to be driven by the 2nd process. You would set up for fuzzing using `prepInject` as if the 1st process were the target. Then provide the `afl` command with the optional `target` and `targetFD` parameters to name the 2nd process and FD from which it reads the data.

6.8 Thread isolation

Sometimes some other thread is in a poorly constructed receive loop, generating lots of breakpoints and slowing down execution. Unless otherwise directed, the `afl` function causes the coverage breakpoints to be on physical addresses, which are shared amongst threads of a process – or amongst all processes sharing a library that is being tracked. Given the costs of tracking context, often this is the best alternative. You can record the basic blocks hit by any other threads, and subsequently keep those blocks from generating breakpoints using the `dead=True` parameter (the `-d` flag on `runAFL`). This causes the coverage function to generate a `[snap].dead` file in the working directory, where `snap` is the name of the snapshot from which the session began. This may take several minutes to complete. When it has found no new blocks hit by other threads for 2 minutes, it stores the dead zones and exits. When starting a new `afl` session, if such a file exists, it will be used to ignore selected basic blocks when establishing code coverage. One symptom of problems with thread isolation is that AFL reports poor stability.

If performance appears to drop way off after a while, consider running with the dead switch again, this time using found paths instead of your seed. Of course this situation could suggest discovery of input data that causes other threads to consume fuzzed data, i.e., yet another shiny object. It would be nice if RESim could provide notice that some other thread is consuming breakpoints, however the act of reading modeled state during a breakpoint slows down the execution cycles by multiples. One clue may be the stability displayed by AFL. A low value can suggest interference by other threads.

6.9 Crash analysis

If AFL finds crashes, use the `crashReport` command to generate reports on each crash. These reports include stack traces, reverse data tracking and indications of ROP or SEGV occurrences. A single command will generate reports on all crashes under a `afl-output` target directory. If the fuzzing inputs potentially include multiple packets, use the `crashReport` utility (not from the Simics prompt) and provide the target FD in the `-f` option. This will use a driver that is assumed to be listening on local port 4022. The ini file must identify the target IP address and port number in the `TARGET_IP` and `TARGET_PORT` ENV variables. The ini file should not name the snapshot created with `prepInject`, it should name a snapshot reflecting state prior to receipt of test data because the driver will send the data rather than having data injected directly into memory.

The original fuzzed data for most recent crash in any Simics instance is written to a file in the workspace called `icrashed`. This data can be injected using `injectIO` (ensure the original filter is applied within the ini file.)

Note: Crash analysis and fuzzing have different criteria for handling `BACK_STOP_CYCLES`. It may be necessary to increase this value to allow time for the crash analysis to encounter the crash.

6.10 Update code coverage

The new code paths discovered by AFL can be added to your aggregate code coverage file using the `playAFL` command (or the `runPlay` utility. This is intended to aid manual analysis to see if the new path leads to other interesting paths that can then be fed back to AFL. The `aflBNT` command displays the branches not taken (BNT). You can then use the `bbAFL` command to identify AFL-generated data files that lead to source of the BNT. Also, refer to the BNT window on the IDA client.

TBD: process for manually eliminating a BNT based on analysis, e.g., no xref to function that modifies prerequisite value.

6.11 Fuzzing performance

The number of breakpoints hit for coverage has a large effect on performance. Avoid references to the target model from within the coverage HAP – a call to `getPID` reduces execution iterations from 20/sec to 5/sec. And see Thread isolation above.

The default is for the `afl` command to set breakpoints on physical addresses. Use `linear=True` to force linear addresses, which will result in use of the `contextManager` to maintain execution context. Depending on the application, this may have different performance properties than use of physical addresses.

One thing that may affect performance when using physical addresses is paging. If a significant amount of application code is not paged in at the start, RESim must dynamically break on page table structures and dynamically add breakpoints and Haps in order to track the basic blocks. Depending on the application and your goals, consider running the program to a steady state prior to creating the snapshot for fuzzing.

Use the `-n` option to the `resim` command to suppress the GUI and windows, this can be the difference between 2 sessions per second and 10.

The `BACK_STOP_CYCLES` value might cause AFL sessions to run longer than needed. On the other hand, too small a value may cause some execution paths to be missed.

When using the `UDP_HEADER` to fuzz with multiple packets, AFL may run amuck, generating lots of packets because it observes what it considers to be unique paths or unique crashes. Consider putting a cap on the number of packets with the `AFL_MAX_PACKETS` ENV value.

6.12 Why fuzz with full system simulation?

Fuzzing a binary often requires the binary to be led to a state in which it is prepared to consume fuzzed data. A general fuzzer such as AFL makes no provision for bringing the binary to such a state. AFL is designed to exec the target program at the start of each session. Getting a binary into the desired starting state so that it can be fuzzed is sometimes accomplished using a custom test harness designed for that binary.

Creating a test harness to fuzz a non-trivial binary can be error prone and time consuming. Instead of constructing a harness, a full system simulator allows you to commence interaction with a target at the locus of execution of your choosing.

Applications that require specific interaction with external entities can be satisfied and checkpointed such that fuzzing sessions start precisely when/where the application is about to consume fuzzed data. Consider a thread that processes selected data provided by another thread that receives network data and writes some version of the data to a pipe.

Fuzzed data is injected directly into application memory, bypassing data reception by the kernel. This can often be achieved with multi-packet UDP or TCP sessions by simply skipping over receive system calls when injecting the next data packet.

The fuzzer, e.g., AFL, relies on feedback from instrumentation on the target to tell it which code paths were hit while consuming a specific input. So long as that feedback indicates new code paths, the fuzzer will pursue those paths in attempts to find more branches on those paths. Sometimes however, we would like to avoid some paths, e.g., to focus on some areas of interest. If the inputs leading to different paths are well enough understood, applying filters to fuzzed data can deter the fuzzer from following unwanted paths. However, sometimes the inputs leading to unwanted paths is not known. While theorem satisfaction tools might help identify such inputs, it may be simpler to simply break on them and feedback a poor coverage report.

6.13 Parallel fuzzing

AFL includes support for parallel fuzzing in which multiple instances of AFL all fuzz the same binary. RESim builds on AFL's parallel fuzzing, and supports deployments having multiple computers, each running several instances of AFL/RESim pairs.

Assuming you have (and are currently in) a RESim workspace directory that contains an ini file and checkpoint, use the `clonewd.sh` command to create some number of copies of the workspace. For example:

```
clonewd.sh 6
```

will create six numbered subdirectories with a prefix of `resim_`. Each of those subdirectories is a RESim workspace having its own `logs/` subdirectory, all other files are simply links.

Then use the `runAFL` command, naming the ini file to be used in the fuzzing sessions. The system will spawn multiple pairs of AFL and RESim, and create an AFL xterm for each to display the AFL status screen. Simics is started without the gui or windows. Pressing **Enter** will terminate all sessions, (the AFL Xterminals will linger for ten seconds.)

AFL results are stored relative to your `AFL_DATA` environment variable (defined in the `.bashrc`) in subdirectories named by the workspace, the hostname and the instances, e.g.,

```
$AFL_OUTPUT/myworkspace/rb9_resim_1
```

The stdout and stderr from each of the Simics sessions is currently dumped to a file at `/tmp/resim.log`. Logs from the `runAFL` command are found in `/tmp/runAFL.log`

Use the `runPlay` command to replay the inputs found by AFL in parallel.

6.13.1 Fuzzing with multiple computers

Multi-computer fuzzing is achieved using scripts that copy your local RESim workspace to a set of *drone* computers, and execute `runAFL` on each drone. This is initiated and managed from the **master** RESim instance, i.e., where your workspace is located. Results from each drone are periodically retrieved and distributed to each other drone.

Create a `drones.txt` file within the workspace directory, containing a list of drone computer host names. Each must be reachable via SSH without passwords, e.g., by use of an SSH Agent. Each drone must be provisioned with Simics and have an account with the user ID used on the master, which must be the host containing the RESim workspace.

- Use the `sync-resim.sh` command to push the local copies of RESim and AFL to the drones and ensure their license managers are running.
- Push copies of your workspace to the drones using the `sync-drones.sh` script.
- Start the multi-computer fuzzing session with `start-drones.py`, passing in the ini file name.
- Pressing **Enter** will stop all fuzzing sessions.
- Get status using `status-drones.sh`

- Kill all Simics and AFL processes with `sync-kill.sh` (usually not needed, but stuff happens.)
- Remove all output data and seeds (for this workspace) from drones with `clean-drones.sh`

The `runAFL` command monitors free memory on each computer and will restart RESim if memory gets too low due to memory leaks in Simics. The AFL sessions persist during these restarts so that they do not lose context. **NOTE:** restarts of RESim inherit the current python scripts. Those should not be edited on hosts having long-running fuzzing sessions.

6.14 False paths

Use the `dedupCoverage.py` utility to identify unique coverage sets. If you find massive reductions, e.g., AFL finds 200 paths and deduping yields four, it is worth going back and looking if AFL is observing extraneous processing, either in another thread or crude read loops. TBD if the latter, add switch to force ending at next read? However that precludes byte-at-a-time receives.

7 Example workflows

7.1 Watch consumption of a UDP packet

Open a pcap with Wireshark. Select the data of the packet, right click and export the selected packet bytes into your simics workspace. Start the monitor and map the desired port to a real ethernet device. Debug the desired process/pid, e.g., `@cgc.debugPidGroup(875)`. Track the IO, e.g., `@cgc.trackIO(14)`, then cat the packet, e.g.,

```
cat mypacket > /dev/udp/127.0.0.1/60005
```

Start IDA with the desired program. Attached to the process and run the `rev.py` plugin. Then go to the “data watch” window and refresh. That will list all instances of references to the content of the UDP packet.

7.2 Reverse engineer a service

This example assumes you want to understand how a program consumes data on a specific TCP port.

- Run the monitor on a configuration having the target and a driver.
- Use `traceAll` to generate a system call trace and the `showProcTrace()` to generate a process trace file.
- Use the `postscripts/genRpt.sh` to generate trace reports, and look at the resulting network information noting the program that binds to a port of interest.
- Create a simple python client script and use `script-driver.sh` to copy that to the driver and run it in the background when the driver boots. This client should connect to the target port and perhaps send a string and read the response. Use a connect loop so the program does not die if the first connections fail due to the target not yet being ready.
- Restart RESim and `debugProc` the target program.
- Use `runToBind` and observe the FD (either in the log or using `idaMessage`)
- Use `runToAccept`, which will not return until the client connects to the service. Observe the resulting FD.
- Use `trackIO` to note where input is processed. The stack may reflect a call to `clib` from some other library. Note you may be in a thread that started in that library. Use `showSOMap` to determine the address at which the library is loaded.
- Open the program or the library file in IDA and use the `dumpFuns.py` script to generate a function map for the library using its initial base addresses.
- If a library was loaded, use `edit/segments/rebase` to rebase the program from the load address observed via `showSOMap`.
- Use `retrack` (or `injectIO`) to re-run the data tracking to pick up calls to mem functions such as `memcpy`.
- Attach IDA’s debugger to the service and work through the `dataWatch` list of Watch Marks.

- Modify data to alter execution paths using either `modifyMemory` followed by `retrack`, or the `injectIO` function (the latter is only available as a RESim command.) The `injectIO` command is most efficient and allows you to alter inputs and see the results without restarting the simulation or restarting IDA (other than to switch libraries.)
- Use the `mapCoverage` function to inventory which basic blocks are hit, and to inventory "branches not taken."
- Stack traces displayed in IDA may lack function names – and may even lack disassembly – depending on the program or library loaded into IDA. You can often still get a decent stack trace using the `stackTrace` command in RESim.

7.3 Observe changes in outputs

Use `traceAll` followed by `traceFD` to observe program output in response to varying inputs injected via `injectIO` with `stay=True`. TBD, combine commands or make modal? – for now, must repeat: `injectIO`; `traceAll`; `traceFD`. This can be more efficient and less complicated than trying to alter inputs of a client and then observing responses from the server.

7.4 Track buffer accesses

You've found a buffer populated with data, and you'd like to track access to the buffer just like you track IO. Use the IDA "add data watch" to add the buffer, and then use `retrack`.

7.5 Find code divergence

Imagine you've found two similar data inputs, e.g., as the result of fuzzing, and one crashes and the other does not. You'd like to know at what point the program execution diverges.

- Use `injectIO` to run each of the sessions, providing the `instruct_trace=True` flag. Provide the PID if tracing is to be limited to a single thread.
- Use the `traceDiff.py` utility with the `-d` option to find where the two traces diverge.
- Rerun one of the `injectIO` sessions, but this time precede that command with the `doBreak` command. Note `traceDiff` tells you if the divergence happens on the `nth` execution of the identified instruction. Provide that value to `doBreak` so that it breaks just before the divergence.
- Use IDA to observe the condition of the branch.

7.6 Branches not taken

See 5.9.3 and 5.9.4 for workflows related to finding branches not taken.

7.7 Packaged example with public images

See `RESim/simics/examples/network_file_system` for an example use of RESim, including IO tracking and fuzzing. The `README.md` in that directory provides instructions for running the example, and for obtaining the disk images and binary files used within the example.

8 Example targets

8.1 Cyber Grand Challenge Services

Vulnerable services from the DARPA Cyber Grand Challenge are available as described in the `README` in `simics/examples/network_file_system` in your repo. The images described therein are located at: <https://nps.box.com/s/ffuz7fgyn770xcgrdur0uf1b01tur2gk>

9 Implementation strategy

This section discusses our approach to implementing RESim, and some implications for the analyst. RESim primarily gathers information about a system through monitoring of events, i.e., observed via callbacks (*HAPs* in Simics parlance), tied to breakpoints. Two key features of RESim enable its flexibility and performance.

1. Other than basic task record structures, the implementation has very little knowledge of kernel internals. This is a key design goal.
2. RESim only monitors events when directed to do so.

Implications of these design properties can be seen by considering example sessions. Assume you boot a system in RESim and let it run a bit without directing any analysis. The `tasks` directive will list currently running tasks. However, RESim would have no knowledge of full program names and arguments provided to `execve`. In this example, directing RESim to debug a currently running PID results in a debug session with limited stack traces because it would not have information from the ELF header⁶ or shared object map information. It would not have information about open files. That information would be collected if the `traceAll` directive were used, or, for a single program, the `debugProc` directive were used prior to the process start.

RESim maintains information it has gathered, and does so across debug sessions and across checkpoints written via `writeConfig`. For example, if you use `debugProc` to isolate a program, and then stop debugging that program and then return to it, the shared object information is maintained.

Other than IDA analysis, we do not maintain state across different sessions other than state used by the `writeConfig` and `RUN_FROM_SNAP` snapshot directives. For example, the shared object maps within two different sessions may vary due to aslr. Simics includes some support for recording and replaying identical sessions. RESim does utilize those.

As can be seen in the implementation scripts (starting with `simics/monitorCore/genMonitor.py`, RESim relies extensively on breakpoints and callbacks (HAPs). Those are the basic building blocks on which most of the functions are implemented. Most of the HAPs are managed by the `genContextManager.py` module. That module also keeps track of processes selected by the user for observation, e.g., via `debugProc`. It is where process death is detected. That module manages two Simics contexts, the default context associated with the cell of the processor, and a distinct `resim` context associated with processes selected by the user. These contexts are how RESim can ignore events occurring in other processes. For example, while debugging a process, you may wish to run forward until that process reads from a given FD, and you don't want to be bothered by other processes reading from the same number.

Analysis during reverse execution occurs primarily within the `reverseToCall` and the `findKernelWrite` modules. The implementations appear (and are) convoluted as a result of how Simics implements reverse execution (and due to limitations of the implementer!). The fundamental issue is that while running backwards, HAPs may be invoked in non-deterministic ways, and thus they must be removed before reversing. The `Core.Simulation.Stopped` HAPs are used to detect when a reverse has hit a breakpoint (or the beginning of the recording). Those HAPs then must determine if the desired event was reached going backwards, and if not, continue backwards.

Details of the implementation and the use of HAPs can be found in this paper: <https://www.sciencedirect.com/science/article/pii/S1742287618301920>

10 Troubleshooting

RESim is an ongoing development and has a limited regression testing system. So there will be bugs. Check the logs, in `workspace/logs`. RESim bugs, e.g., Python errors, are often masked when driving from the IDA Client. When things seem broken, they may well be. Redo what fails using the RESim command line, and you may see the error.

IDA Pro's debugger at times may get lost. You can exit IDA and restart it using the `runIDA.sh` command followed by the `shift-r` hotkey to reconnect to the simulation.

If you find yourself in the kernel, you can usually use the `run2User` command or its IDA debugger equivalent.

The simulation may reach a state that you cannot proceed from. Use bookmarks (or watch marks) to return to a known state.

Information about share object file use by processes is only gathered when directed by use of the `debugProc` command. While you can attach to any process at any time, this key shared object information will not be available. See 9 for more information.

If you observe divergence between different runs of what should be identical simulations, consider whether the real world is leaking into your simulation in any way – see section S.1.

⁶Unless used with the IDA client

Double-check your ini file snapshot. Recall there are often 2 snapshots used for any given target; one for data injection and one for driver-driven data.

If you observe non-repeatable crashes, e.g., SEGV, confirm you are not injecting data past the end of a read buffer.

If RESim appears to no longer run, e.g., runAFL just hangs, check that there are no zombie simics processes running. Use the `kill-simics.sh` script to kill them (but do not kill the `lmgrd` instance).

Reverse track of data might go off the rails for a variety of reasons. When this happens, restart and try reverse-wrote-to-address

If an AFL session seems to hang, stopping it will cause the most recent AFL data to be written to the `last_afl.io` file, which can then be analyzed. instead of the full tracking.

11 Data Stores

This section describes how RESim stores and accesses different data stores.

One goal is to simplify sharing between the RESim platform and the IDA platform, which may be the same, or different.

IDA and RESim need to share binary files; function and block lists; and hit lists.

To avoid NFS conflicts, only read-only files should be on NFS. We trade this off against the desire to share basic data such as IDA's functions and blocks lists.

The binary and the `.funs` and `.blocks` files will be stored in the at locations defined by the `RESIM_ROOT_PREFIX` value in RESim ini files. The intent is for these to be on an NFS share so that the same ini file can be used on multiple RESim platforms.

The `idb` and `hits` files will be per-user, stored in a local writable directory relative to path named by the `RESIM_IDA_DATA` environment variable. Files will be copied between IDA and RESim platforms using `scp`. Each analyzed program has a subdirectory in the `RESIM_IDA_DATA` directory. Within that subdirectory is the IDA `idb` file and the `hits` file. It also contains a file with a `".prog"` extension that identifies the path to program that was analyzed.

The `hits` files are stored as lists of basic block addresses. When `playAFL` is run the `hits` file includes the name of the target. Per session `hits` files are also stored in the AFL output directory. These are dictionaries keyed with the basic block address and the values are the cpu cycle at which the hit occurs.

The addresses in `hits` files will be based on the static program addresses, i.e., not adjusted by the load offset. Other data sets that include instruction pointers, e.g., `trackio` records, should include SO map information so that their instruction pointer values can be normalized.

12 Utility scripts

The following is a summary of a set of utility scripts in the `RESim/simics/bin` directory. These are typically run from a bash shell in the workspace directory.

- `runAFL` – Start AFL sessions for a given target.
- `clonewd` – Create a given number of cloned workspace subdirectories to run AFL/RESim in parallel.
- `start-drones` – Start a set of RESim/AFL pairs locally and on remote computers named in the local `drones.txt` file. See [6.13.1](#)
- `stop-drones`
- `sync-resim` – Push RESim code to the list of drones in `drones.txt`
- `sync-drones` – Push the current workspace to the list of drones.
- `status-drones` – Get AFL status of the drones.
- `runPlay` – Replay all path inputs generated by AFL. Will run in parallel based on the number of clones created with `clonewd`.
- `crashReport` – Run automated analysis on a set of crash files found by AFL. When inputs may include multiple packets, the utility will send packets via the driver computer which is assumed to have port forwarding on port 4022. Use the `-f` option to identify the FD.
- `runTrack` – Use the `injectIO` command to generate `trackio json` files for sessions found by AFL. This restarts Simics for each session to avoid redefinitions of the origin that would occur if `trackIO` were used multiple times in a single Simics session.

- `findBNT` – find branches not taken within the program hits file in the `RESIM_IDA_DATA` directory.
- `findBB` – List the AFL session files that reached a given basic block.
- `rmLogs` – Remove log files from AFL clones within a workspace (assumes you are in the workspace directory.)
- `showCoverage` – Show basic block coverage of an AFL file, or all files.
- `dedupeCoverage` – Identify AFL sessions that resulted in unique sets of hit basic blocks.
- `dataDiff` – Experimental, compare trackio output generated from post-AFL processing.
- `traceDiff` – Compare 2 instruction trace files and locate points of divergence (see the `instructTrace` command).
- `diffHits` – Show differences between 2 hits files.
- `idaDiff` – Compare two coverage hits files in IDA, using basic block coloring to highlight differences. Use `diffHits` to find points of divergence. The hits files must be in the `$RESIM_IDA_DATA/program` directory.
- `cycleSeeds` – Populate the AFL seed directory with queue files and manually added files identified in the results of the `dedupeCoverage` utility.
- `kill-simics` – Sometimes it is the only way.
- `drive-driver.py` – Send data from a driver computer to a target IP/PORT. A real network is used to push the data and scripts to the driver. There are many ways to send data to a target computer, and this is an example. See the script for syntax of its directives.
-

13 Testing

Many of the RESim functions are exercised by automated tests found in the `simics/testing/cadet-test` directory. These include `trackIO`, `injectIO` and fuzzing with AFL. The tests create a new RESim workspace relative to the directory from which the tests are run. Use the

```
$RESIM_DIR/simics/testing/cadet-test/tstcadet.sh
```

command to initiate the testing. The tests require the `xdotool` package.

Appendices

Appendix 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 must 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 heuristics and brute force to find the remaining parameters.

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.

B.1 Faults on ARM

The x86 case seems simple compared to what is found in ARM, whose exceptions include “Data Abort”; “Prefetch Abort”; and “Undefined Instruction”. Data references to unmapped pages yield a Data Abort; while instruction fetches yield a Prefetch Abort. The “Undefined Instruction” is not necessarily fatal – for example we see the `vmrs` (some floating point transfer) a lot.

Data Aborts lead to page handling, unless it does not. Of interest is that it can lead to references from the kernel to addresses provided by user space.

B.2 In process

For a while, both ARM and x86 used instruction breakpoints to catch faults. But we still must enable `Core_Exception` breakpoints to catch illegal instructions. ARM prefetch abort entry for 64-bit cpu is 0xffff0280.

How to find that with kernel parameter harvesting? Maybe set `Core_Exception` haps and step/record? We've subsequently gone back to only relying on `Core_Exception` breakpoints.

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. Use the `showSOMap` command to view a list of libraries and their load addresses. The `stackTrace` command identifies the library of the current stack frame (or `show` for the current instruction address. Once you know the library and its load address, open the library in IDA (and use `dumpFuns.py` and `findBlocks.py` to generate databases referenced by RESim if not yet created). Rebase the program (`Edit/Segment/rebase` and then attach to the debugger. NOTE, depending on the library, rebasing may take a while. Wait for IDA to finish before attempting to attach the debugger. Switching between libraries currently requires that you exit IDA and then open the other library.

Appendix 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 system calls are tracked and compared to a frequency threshold. When it appears that the program is spinning on a clock or an event such as `waitpid`, 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. This feature is called a *maze exit*. If you would like to avoid the prompts, use the `@cgc.autoMaze()` function to cause the system to automatically try to exit mazes as efficiently as it can.

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.)

This feature is currently only functional on X86 platforms.

Appendix 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 mechanism 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. It happens that while 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 occur*. RESim takes advantage of the kernel’s 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 that we can read.

Note this property of the 64-bit Linux implementation 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.

Appendix 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 5.5.)

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. Models may lack specific peripheral devices expected by a kernel. In some cases, functionally compatible devices can be substituted for missing peripherals. For example, the kernel image from an X86 target may include drivers for existing Simics ethernet devices, and yet the initialization functions do not cause the corresponding modules to be loaded – rather, they load kernel modules for an ethernet device that is not modeled. Use of the Dmod function described in 5.5 can cause the kernel to load the desired module, without having to modify the disk image. TBD: Explore model building, e.g., to simulate an fpga accessed via a PCI bus device.

F.3.1 VLANs

While Simics has support for VLAN switches, the network interface models do not support VLAN definitions in which the VLAN subnet differs from that of the parent interface, and the model requires the parent interface to be defined. Wind River considers this common configuration of VLANs (with differing subnets) to be a “use case” beyond their intended features.

F.3.2 File descriptors

File descriptors (FDs) can diverge between boots of some systems in ways that seem non-deterministic. Additionally, divergence between modeled hardware and real hardware can contribute to FD divergence. For example, if a model lacks a peripheral device cause an `open` to fail, subsequent FD assignments on the model will not match those used on the physical system.

Appendix G What FD is this?

You have created an input stimulus and would like to understand the resulting behavior of a specific process that runs a long time as a service. Running a full system trace up to the stimulus as a means of getting context may not be practical. So you start a `traceAll` at some point prior to the external stimulus. You then see socket activity on a specific FD. However the current trace does not include a connection that maps to that FD. Use the `procTrace.txt` file to get the pid of the process as it existed in the full system trace. Then, if you assume the FD will match that from your full system trace (which may be a fine assumption if the connection is long-term), `grep` on the system call trace, e.g.,

```
grep "pid:1731" syscall_trace.txt | grep "FD: 11"
```

That gives you the port and address information that you can then search for in the `netLinks.txt` file.

Appendix H Context management implementation notes

RESim manages two Simics contexts: the default for the cell, and a “resim” context for each cell. Contexts are managed within `genContextManager.py`. The resim context is used when watching a specific process or thread family. Otherwise, the default context is used. The context of each processor is dynamically altered such that breakpoints are only hit when in the corresponding context. For example, assume we are debugging PID 95 (for simplicity, assume no threads). Whenever PID 95 is scheduled, the processor context is set to resim. The context is returned to the default whenever PID 95 is not scheduled. System call breakpoints set during debugging, e.g., `runToWrite`, will be associated with the resim context. These breakpoints will only be hit when the processor is in the resim context. Breakpoints may be set in the default context while debugging, e.g., `handle a background Dmod` (see 5.5.2). Those breakpoints will not be caught when processor is in the resim context. TBD: define multiple contexts to allow parallel debugging of different processes on the same cell. (Parallel debugging of processes on different cells should be easier since they each have independent context managers.)

IDA breakpoints and the Simics debug server make assumptions about the current context. If you are in an arbitrary state, you will need to reestablish the debug context before IDA breakpoints will work. Use the `cgc.resynch()` or `Run to user` in IDA to run the simulation ahead to the debug context.

Appendix I What is different from Simics?

RESim is based the Simics simulator, including the *Hindsight* product. It does not incorporate the *Analyzer* product OS-Awareness functions. 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.

Appendix J IDA Pro issues and work arounds

The IDA debugger in 7.2 requests more registers than Simics knows about. The Simics gdb-remote has been modified to simply return the value of the highest numbered register that it knows about. An alternative is to modify xml files in `/ida-7.2/cfg` to only include desired registers. For ARM (the qsp-arm), the `arm-fpa.xml` (from gdb source tree) needs to be added to the `arm-with-neon.xml` file in `ida/cfg`. The `cpsr` register (number 25) is not updated by IDA unless registers are defined for each register value returned by Simics in the 'g' packet.

The IDA debugger in 7.2 uses thread ID 1 when performing the vCont GDB operation. Hex-rays modified the IDA gdb plugin to use ID 0, thereby fixing the problem.

The block coloring in IDA fails sometimes, possibly when the function includes unmapped pages. The coloring works fine prior to attaching the debugger. After attaching the debugger, some blocks revert to their default color. Use the Debugger/RESim/Recolor Blocks menu function to recolor the blocks. This will usually correct the problem. But by that time you already know something is messed up.

Appendix K Simics issues and work arounds

- New-style Simics console management causes an X11 error: "The program 'simics-common' received an X Window System error." Use:

```
gsettings set com.canonical.desktop.interface scrollbar-mode normal
```

to avoid the exception.

- The ARMv5 model had a flaw that reports in incorrect fault status on some data aborts. Wind River has now issued a patched release of the Integrator (6.0.12).

If you lack the patch, see the `fixFaults` function in the `afl.py` module.

- The service nodes used for real-network interaction were not saving their ephemeral port numbers on write-configuration. As a result, the server could see reused port numbers, causing the TCP connection to fail after a restore from the snapshot. Wind River has fixed this. We currently have a patch that will be included in the next Simics 6 release in mid 2022.

Appendix L Performance tricks

Use `disable-reverse-execution` whenever you do not really need reversing, e.g., while moving forward to a known connect or accept within a program being debugged. Actually, use `cgcc.noReverse()` and `cgcc.allowReverse()`, which will disable the sysenter Haps maintained by the `revToCall` module. Programs can be pathological with their use of syscalls.

Instead of tracing all from a boot, consider running ahead until the `systemd-logind` or `rsyslogd` is created.

Pay attention to the `BACK.STOP.CYCLES` environment variable in the ini. A large value may be needed during initial testing, but that may need to be greatly reduced for fuzzing.

Some system may take a long time to create the process you are interested in. This may be exacerbated by monitoring such as occurs during a `runTo` or `debugProc` (e.g., some script is in a loop failing on create of a process, which may happen millions of times.) Consider using `runToCycle` to run forward to a desired cpu cycle with the least amount of overhead.

Appendix M New disk images

For x86, use `put the workspace/dvd.simics` in targets. Edit it to name an installation iso. You may need to change the date in the file, and run with `realtime` enabled. After the install, use `save-persistent-state`, and then rename the resulting `crash` as needed.

When using real networks, configure the image routing and resolv.conf, using your ip address:

```
route add default gw 10.10.0.1
echo nameserver 10.10.0.1 > /etc/resolv.conf
```

Appendix N Implementation notes

Use the `VT.in_time_order(self.vt_handler, param)` when you need to know the memory transaction that led to a stop hap during reverse. See `findKernelWrite` for an example.

Take care when using `SIM_hap.add_callback_range` to ensure your breakpoints are truly in a contiguous range. Concurrency often results in dis-contiguous allocations of breakpoint numbers with the "holes" subsequently used for other purposes. Imagine your confusion when a hap is hit for the wrong event. If needed, issue multiple haps on multiple ranges, watching the breakpoint sequence numbers as they are allocated. Simics breakpoint numbers may look like handles, but the hap range allocation gives their values semantics. See the `coverage.py` module for an example.

HAPs are often hit after they are deleted. Always set a deleted HAP to None and check that at the start of the HAP. Deleting a breakpoint does not stop its HAP from being called when the deleted breakpoint's address is hit. As such, optimization schemes should not try to reduce breakpoint counts on the fly.

Appendix O ToDo

O.1 Msc

Trace params of `pctl` and catch `setname`, which is used by `systemd` to change thread name to `rsyslog`. Should be able to break on this like we do for `execve`.

Catching kill of process uses breakpoints on task record next field that points to the subject process's task rec. Fails if the process whose record is watched is killed. Need to also watch "prev" on the subject process?

Convert traces to csv and explore data presentation strategies for navigating processes and IPC.

Feature to flexibly identify user-space libraries to be traced.

Enhance the `dataWatch` function improve detection of generic C++ string allocator, and expand watch.

Tracking memory-mapped IO is not directly supported. Perhaps catch the `mmap` function call and somehow determine it is mmio purposed. Then set breakpoints...

`sscanf` data tracking not right for some arm, if 1 or 2 values, `r2` and `r3` will point to where those go rather than using a pointer list.

The `playAFL` function could disable reverse execution, also have it set jumpers to not clear bookmarks.

Confirm `crashReport` uses jumpers.

Add read counts to data watches.

`dMod scriptReplace` sets an `lseek` `DMod` to adjust future `lseek` calls. The `dmod` sticks around until the file is closed (even then?, would have to track `pid/FD`).

Add fuzzing/coverage option to instrument main program and a lib(s).

IDA hotkey to go to address adjusted by the start of the `LOAD` segment.

Ad-hoc copies in `dataWatch` not robust for arm, ignoring register overwrite until we can eliminate case where register is overwritten with a value derived from its value (x86 has same robustness issue, but it is not ignored).

Investigate ways of limiting IDA debug data sent between client and Simics. Causes slow responses on remote debugs?

Extend `maze exit` to ARM.

Error handling when running AFL. Filter failures cause a hang with no messages. Simics crashes sometimes seen in `/tmp/resim.log`

`writeData` has no fallback for a page fault. Add a hap on the page table to complete the write? NO, force big `prepInject` data

Use of `debugSnap` after having run a while loses state information about threads waiting in syscalls. Add warnings if simulation is continued without debugging if there is a snapshot. Use hap that catches `continue`?

O.2 I/O via threads

Network traffic is sometimes sent via a thread, making it challenging to track the source of the data that was sent. Breaking on a "sendto" may lead to a thread that only does the `sendto`, with the parameters coming from a clone setup. Thus, you must find which `pid` does the send to and then run `ToSyscall` until that `pid` is created.

O.3 Tracing library calls

It should be straight forward to instrument selected relocatable functions or statically linked functions. May be tempting to do that for malloc – but on the other hand, if you have the address of the start of a buffer, then reverse tracking that will generally lead to the malloc call.

O.4 Watching process exit whilst jumping around time

We try to catch page faults, or other events that lead to process death. This is performed in the ContextManager. This mechanism is also central to catching access violations. The mechanism works fine moving forward from a clean state. But how does it behave if we jump backward to an arbitrary time, e.g., prior to the demise of a now dead process? A new ContextManager function will reset all process state to that currently observed. TBD, also modify tracers to ignore events that occur prior to end of recording?

O.5 Defining new targets

System names are defined by assigning "name = whatever" within the ...system.include file's line that creates the component, typically the board. Simics parameters are typically ONLY added to the script file in which they are used, and then sucked in by the calling scripts using params from. Systems scripts error reporting is sparse and tedious. Copy the needed Simics scripts from their distribution directory into simics/simicsScripts/targets, and change the simics env variable to scripts where needed – or remove preface and use relative file if local.

O.6 Fuzzing: too many crashes

Some applications are just too easy to crash. This can result in scores of “unique” crashes from AFL. Often, these are all the same crash with a small change in how we get there. Any heuristic that suppresses a crash seems like a bad idea, unless analysis convincingly demonstrates otherwise. Perhaps a tool to scan crash reports to eliminate duplicates, i.e., we usually don't care how we got there if a high confidence stack frame is the same? And crashReport function itself should run that tool when done, along with a grep on ROP.

O.7 Branches not taken

The injectToBB function simply finds any fuzzing input that leads to the given basic block, and then injects that data. If the bb includes a select or similar, the execution may never reach the branch logic. Alter the findBNT to filter out those that do not follow any of the branches.

Manually working through BNTs can yield manual inputs that reach new branches, but may always crash. These inputs are not helpful to fuzzing because crashes don't feed the guidance system. These manual inputs can also cause injectToBB to bring up a crashing input even though non-crashing inputs might yield the same bb. More reason to keep crashing inputs out of the manual batch. Create a new manual crashes? Manual crashes would have their own coverage, which would be referenced when running findBNT, but would not be referenced by injectToBB.

Similarly, manual BNT analysis might yield crashes. Do not let that stop you from looking at other inputs leading to the branching BB, i.e., to find inputs that yield new code paths without crashing, since these are helpful to fuzzing and coverage. Crashes are good, but sometimes they are dead ends. So use findBB to locate other inputs besides the one picked by injectToBB.

O.8 Skipping over kernel calls

The data injection scheme catches kernel reads and skips over them while injecting data. This affects timing/scheduling because the kernel never blocks.

O.9 Data Watch ntohs

The ntohs function is common in some programs. Consider treating it results as a data copy if easy to find where in memory it lands. e.g.,

```
call    _ntohl
0808A7E0 add    esp, 10h
0808A7E3 mov    edx, eax
0808A7E5 mov    eax, [ebp+arg_0]
0808A7E8 mov    [eax+68h], edx
```

O.10 More on kernel data buffers

On a 32 bit x86, there were many 14xx sized kernel buffers. A new option for trackIO (used prior to calling prepInjectWatch) was created for recording these buffers. This option, kbuf=True, requires drive-driver.sh to send as large a data stream as you wish to support for future injectIO or AFL sessions, and this data stream must be crafted so that the application reads all of it. A hap is set on the read/recv call, and this hap sets a write break on the start of the application buffer. When that is hit, we assume that esi contains the start of the kernel buffer, and it is recorded. A hap is also set on read returns. The consumption of the kernel buffer is monitored along with the amount of data read by the application such that before a kernel buffer is consumed, a write break is set on the next application memory location that will receive the first byte from the next kernel buffer.

And fuzzing design in general... In addition to basic problems that can be solved by filters, sometimes fuzzed data does not conform to system state values. For example, previously received data may contain a crc and/or length of the data to be fuzzed. Jumpers around crc calculations and memory modification should be avoided simply because it requires code analysis to determine completeness, e.g., is that the only reference to the predetermined code length? In general, it is preferable to start the fuzzing at the inject of the earlier data, e.g., by using a filter.

O.11 Data watch ad-hoc copies

Treated as an ad-hoc copy?:

```
mov x, buf
push x
call foo
mov bar, x
```

O.12 UDP without headers

The injectIO and AFL design for UDP relies on their being UDP headers to break data into multiple packets.

O.13 TCP conversations and fuzzing

Consider a case of fuzzing TCP. Assume the target is a client and the driver is a server. The driver component is directed to run a server that accepts on the port and sends a given file of data.

You use the kbuf option to trackIO and then prepInjectWatch which gives you snapshot on the call to recv with lots of data in the kernel buffers.

Fuzzing yields the start of a conversation, i.e., where the target makes a 2nd call to read (or poll/select). It is assumed the first call to read had a length larger than the count. (Find a way to distinguish a conversation from just reading more data. For example, one or more writes to the socket between the reads.)

We'd like to use this conversation start to advance system state, and then start fuzzing with the next read. Use a multifile driver directive to force a read between the two writes.

Assuming the target is a server, a work flow might look like:

- Create an initial snapshot ("snapshot0") in which the server is ready to accept connections. This snapshot will be the direct parent of all other snapshots. (Once a prepInject snapshot is created, it cannot be reliably advanced.)
- Set the ini file snapshot to snapshot0
- Use trackIO with kbuf=True on a driver directive that sends Zs.
- prepInjectWatch(1, 'snapshot1')
- Set the ini file snapshot to snapshot1
- fuzz (session 1)
- Find a dangling conversation queue file, e.g., the service does a write to the socket after consuming data, and then issues another read (or poll/select).
- Set the ini file back to snapshot0
- Create a directive that starts with that queue file, and is followed Zs, with an intervening read.

- Use trackIO with that directive, which should hit a backstop after processing the first part of the conversation.
- Again use trackIO, this time with kbuf=True
- prepInjectWatch(N, 'snapshot2'), where N is the watch mark of the 2nd kernel read.
- Set the init file to snapshot2
- fuzz (session 2)
- Find the next dangling conversation from fuzz session 1 and repeat
- When all fuzz session 1 conversations have been run, proceed to fuzz session 2 dangling conversations.

We are interested in whether fuzz session 2 reaches code not hit by session 1. There should be some data mapping between the afl output of the fuzz sessions, the snapshots, and the directive file that led to the snapshot. All snapshots descend directly from snapshot0.

O.14 Fuzzing crash detection

While fuzzing, we want to detect and report crashes and/or process exits. User control over these? Currently only report crashes.

Appendix P Driver platforms

This appendix describes the driver image used internally at: `/eems_images/ubuntu_img/driver/driver.disk.hd_image.craff`. The driver is also available at <https://nps.box.com/s/ffuz7fgyn770xcgrdur0uf1bo1tur2gk>

The driver platform defined by the `RESim/simics/workspace/ubuntu_driver.ini` example has the Simics Agent pre-installed. The `driver-script.sh` in that directory can be copied and modified within your workspace. The driver will download that shell script and execute it when booting. Your target will not boot until that script finishes (and creates the `driver-ready.flag` file). Note the simulation will hang while processing that script, so launch any long running programs in the background. The sample driver script loads an `authorized.keys` file from that directory. This allows ssh into the driver without passwords and uses rsa key files also found in that directory.

The username and password for the driver is mike. No password is required if ssh keys are used. Use `enable-real-time-mode` to avoid login timeouts and network timeouts when interacting directly with the driver.

The driver device has 4 ethernet interfaces. The mac addresses below are as defined in the ini file:

- ens11 – 10.20.200.91/24 mac: 00:e1:27:0f:ca:a8
- ens25 – 10.0.0.91/24 mac: 00:19:a0:e1:1c:9f
- ens12 – 172.31.16.91/24 mac: 00:1a:a0:e1:1c:9f
- enp7s0: not defined mac: 00:1a:a0:e1:1c:a0

Note that use of the ethernet 82559 device changes ethernet device names, e.g., ens11 becomes ens11f0 and ens12 becomes ens12f0.

Use `ip addr` commands within the driver script to modify these addresses are required.

The mapping of RESim ethernet names to Linux device names in the driver is:

- eth0 - ens25
- eth1 - ens11
- eth2 - ens12
- eth3 - enp7s0

By default, the ETH directives in the ini file to connect different ethernet devices to the different switches. The defaults are ETH0-to-switch0, etc.

The ini file maps multiple ethernet devices to switch0, which avoids the problem of tracking connections. This works so long as the networking is simple. The following Simics command provides real-network access to this sample driver:

```
connect-real-network 10.20.200.91 switch0
```

Once the driver is booted, you can then ssh to it using:

```
ssh -p 4022 localhost
```

You should first put Simics in real-time mode (`enable-real-time-mode` to avoid timeouts. You may also need to clear out the ssh keys used by localhost, e.g.,:

```
ssh-keygen -f "/home/mike/.ssh/known_hosts" -R [localhost]:4022
```

To update the driver, e.g., with new packages, use the driver.ini configuration (after first adding `eh DRIVER_WAIT` directive. Then use `mapdriver.simics` to connect to the real network. You can then use `apt-get` to get new packages. And `scp` to push files onto the machine. Then use `save-persistent-state` and the `do_merge.sh` to create a merged `crdff`. **Do not** forget to shutdown or sync before saving state!. Give the driver `crdff` a name that does not conflict with existing names.

Large `scp` operations may stall. Consider using the `simics-agent` to move files to the driver. However, these only happen one file at a time, so use `tar`.

Add the `simics-agent` from `RESim/simics/simics-agent` to `/usr/bin`. And create a `systemd` service to run the `driver-init.sh` script from the `RESim/simics/workspace`. That script bootstraps the driver's ability to pull and execute the `driver-script.sh` from the workspace.

Smaller `scp` and `ssh` sessions seem to work well with drivers. Add an `authorized_keys` file to what the `driver-script.sh` to facilitate xfer of scripts/data to the driver during the simulation. Take care to reset the reversing origin after you are done with the real-network (see the `trackIO` reset option.)

P.1 Notes on updating the driver

If building a new driver, use the `DRIVER_WAIT` directive in the ini file to prevent `RESim` from waiting for the driver to finish (the `driver_ready.flag`) – at least until you finish configuring a driver.

The Wind River *real network* interfaces are not entirely stable, as a result, the `real-network` connect used to update the driver can often stall/hang. Be prepared to kill `apt-gets` and retry multiple times. It will eventually work.

The Python world uses SSL certificates in a manner that leads to breakage. If `pip` fails on SSL validation, use the `-v` option to find which new `python.org` server is causing the problem, and use the `--trusted-host dog.pythonwhatever.org` option on `pip`.

Appendix Q Simics user notes

This section includes ad-hoc suggestions for users with little Simics experience.

- Simics documentation is in the `doc` directory relative to workspace directories.
- See the *ethernet...* manual for connecting real networks.
- Use `wireshark <switch>` to see traffic going through one of the switches. Note that `wireshark` in the Simics environment is sensitive to changes made to the simulated system. For example, if `Wireshark` is running then writing to memory will cause `Wireshark` to not report traffic. If you want to watch traffic with `Wireshark` when using the `injectIO` command, use the `stay=True` option, then start `Wireshark` and then continue the simulation.
- The `enable-realtime-mode` prevents the simulation from running faster than real time. Useful when trying to login to a virtual console, or avoid network timeouts. Also useful when working remotely and you cannot get a `ctrl C` in edgewise.
- The `"x"` command at the `simics` command line displays memory values. Use `"help"` and `"apropos"` to find other commands of interest.
- Use `pselect` to control which `cpu` you are viewing.
- The `pregs` command displays registers.
- To save changes made to an OS disk image, use `save-persistent-state`; and then exit `simics` and use the `bin/checkpoint_merge` command to create a new `crdff` file (found in the `checkpoint` directory).
- To trace all instructions and memory access use:

```
load-module trace
new-tracer
trace0.start
```

- The Simics command line uses standard bash history. The up arrow can save much time and syntax problems.
- Capture command line output (redirect a copy) to a file using the `output-file-start` / `output-file-stop` commands (`start-command-line-capture` replaced that).
- `list-port-forwarding-setup` to view current port forwarding

Appendix R Injecting to kernel buffers

TBD integrate all injection sections, and clarify different approaches: `ioctl`; `select` / `read`; x86 buffers

The `injectIO` function is hamstrung if the application makes many single-byte read calls, because writing data into the simulation forces a reset of the origin used for reverse execution. This may also arise when protocols are implented with buffered reads, e.g., 80 bytes at a time.

One solution to this is to inject the data into the kernel buffer instead of into the application buffer. We can then use `trackIO` and not have to write additional data into memory. This strategy has several limitations, but is also suitable for some environments. In general, the goal is to provide the application with a realistic view of input data in the course of its processing. There are two potential strategies for injecting directly into kernel buffers. The first is to intercept and patch return values from calls such as `ioctl` and `select`, and just allow the kernel to return previously buffered data for each read/recv call. The second approach is to modify the kernel's data structure used to track data in the input buffer. Neither approach attempts to fully synthesize the kernel input processing, and thus running forward past the application processing is likely to reflect divergence or kernel errors.

Intercepting kernel calls such as `select` can get complicated, particularly if the application is using timeouts and relies on the intervening blocking. Altering kernel buffer pointers avoids some of those complications. The snapshot created for use by AFL or data tracking must then include the address of the kernel buffer, and the addresses of the two pointer values that the kernel references when responding to read calls. Each are found by using reverse data tracking within the `prepInjectWatch` command, which is intended to be used after the analyst does a `trackIO` and observes the system calls. Current support assumes a call to `ioctl` followed by a read, or use of `select` prior to a read. The snapshot will be made prior to the call to `ioctl`, or prior to the read. RESim back traces the return value from `ioctl` to get the pointer addresses, and it backtraces the read data to get the kernel read buffer. The `WriteData` module then uses these addresses to modify the kernel buffer for each data injection.

Appendix S Troubleshooting

S.0.1 Missing syscalls in logs

Syscall haps for the default context and the RESim context can co-exist. The presence of syscall log entries does not mean that all the syscalls you intend to catch are being caught.

S.1 Real networks: WARNING

Simics supports traffic between the simulation and a real network using `connect-real-network` commands and related commands using a *service node*. This can be useful to quickly generate new packets and send them to the target, but without needing to interact with a driver component. Note however that Simics has limitations on the use of real networks when reverse execution is enabled, or memory shapshots (`restore-snapshot` are used. Some of these limitations can lead to silent corruption of the simulation. Others to crashes. An obvious limitation is that you cannot replay periods in which real network traffic was received (though Simics supports a separate IO replay feature).

Most problems related to real network can be avoided by connecting and completing use of the real network prior to enabling reverse execution, e.g., via a `debugProc` command. Or using `resetOrigin` after all real-network IO is finished. See the `reset` option to the `trackIO` function. It should be used when real-networks interact with drivers, e.g, an ssh script to cause data to be sent to the target.

Automation, such as the `prepInject` function can make use of Simics magic instructions to allow a driver computer to cause a new origin to be established. See the `simics/bin/driver-driver.py` program as an example.

Use the `INIT_SCRIPT ENV` directive in the init file to specify a simics script to load at the start of each session.

Real networks should be avoided when debugging, fuzzing or replaying TCP services since they can lead to undefined behavior.

S.2 Failed cat to /dev/udp/localhost

The name `localhost` fails on the blade servers (old Linux?). For example:

```
cat fu.io > /dev/udp/localhost/6060
```

should be, instead:

```
cat fu.io > /dev/udp/127.0.0.1/6060
```

S.3 Real networks and UDP

Using `cat foo > /dev/udp/localhost/60060` is fine for a single packet. However multiple packets seem to get lost unless brief sleeps are added between the cat commands.

S.4 Backtracing malloc'd addresses

You have the address of a buffer you think was malloc'd; you back trace, which stops in malloc. You think you found it, look at the call stack and declare success. You may be wrong. You may be looking at the malloc that happened prior to the malloc of interest. For example, if you are looking for the source of a memory location whose value is 0xf4938, RESim may find that in malloc, and then happily continue to back trace until it finds the creation of value 0xf4930 – just in increment, right? So, look at your cycles and notice large gaps. Add a "value" field to the bookmark printout to make it more obvious when a permutation on the desired value is being reported.

S.5 Fork exit

A fork followed by an exit may result in the exit occurring before the child is ever scheduled.

S.6 Fuzzing

If the number of paths seems stuck low, look at the cycle backstops. If the application will loop looking for more data, that will lead to a perceived hang. Use the `STOP_ON_READ=True` to cause the session to stop when the application cycles back to the read. This has obvious limitations.

S.7 Fuzzing TCP

Programs may be sensitive to packetizing that occurs under TCP. All testing and operations may have occurred with complete data transfers in which each read call returns all data needed for a transaction. If the fuzzing session were to force the client to perform multiple reads, untested code might be exercised.

S.8 Multipacket workflows

Often, when working with multiple packets, you need to avoid use of real networks and data injection to avoid potential corruption of the simulation. This is generally true when reverse execution is required for the analysis. This is not the case for code coverage operations such as AFL sessions that have no requirement for reverse execution. Analysis over the full session of a multipacket simulation requires a driver computer that sends the desired packets. Any checkpoint used for such a session must reflect time prior to the return from the read/recv call. Sessions used for fuzzing require a snapshot that reflect the precise return from the first read/recv call.

We therefore expect many workflows to use two different snapshots, one for data injection and one for analysis over the full data period, e.g., via trackIO. The latter requires at least two computers in the simulation.

When TCP data is injected into kernel buffers, the state of the kernel becomes corrupted, thus preventing running forward, e.g., to a new checkpoint. In order to change state of the system, revert to a running snapshot and send the crafted data via a driver. Again, do not try to move a `prepInjectWatch` checkpoint forward.

S.9 Your target process is not scheduled in the snapshot

Consider debugging a service that loads before you want to create a snapshot, e.g., you wish to wait for other processes to start. You wish to debug the target's data consumption and you know it will hang on a select or a read. So after `debugProc`, you run a while without giving it data and then create a snapshot. When the snapshot is restored, your target process is not scheduled, and thus the `debugSnap` function will appear to hang until you provide data that causes your target to be scheduled. This is expected.