Contents

1 Using Stackdb

Once you've built and installed Stackdb, you'll probably wonder how to get started. This guide will help you understand Stackdb features and usage model; configure your system to take full advantage of it; give you a brief tour of the programs and tools it comes with; and help you get started writing your own Stackdb programs.

1.1 Quick Start

It's easy to get started with Stackdb and do some debugging quickly, so this section is short. If you're already familiar with Stackdb, just install Stackdb, and jump to the Examples and Demos section.

1.2 Understanding the Concept and Features

Stackdb is not a normal debugger nor memory-forensics tool – although it can serve both those functions very well – so the first step is to familiarize yourself with the features Stackdb offers. You can download and read our paper at http://www.flux.utah.edu/paper/johnson-vee14.

If your interest is in simply using a few features of the system, you'll be able to skim some of the more complicated details; but we do encourage you to read enough to gain a conceptual understanding of Stackdb if you're not familiar with it.

1.3 How to Run Stackdb Programs

The purpose of most Stackdb programs is to attach to a target program – an OS running in a VM; a process running in an OS running in a VM; a process running in userspace; or even a PHP application running in a process running in an OS running in a VM!

Once attached, usually you want to extract some information from the target, either by reading its memory, or detecting and analyzing its control flows at key points.

Each Stackdb program is a peer of the base target it attaches to. That means, if it is attaching to a VM to debug an OS, it runs in a control VM and attaches to the target VM. For instance, if you're using Xen, you'll run your Stackdb analysis/debugging program(s) in Domain 0, Xen's privileged control domain, and attach to other Domains. If you're running QEMU/KVM domains, your Stackdb program(s) will run in the Linux host, and attach to the QEMU process that is running your target VM. If you simply want to attach to a Linux userspace process, just like using GDB to debug a normal program, both the Stackdb program(s) and the userspace process run in userspace, as peer processes. Stackdb processes are just debuggers that you attach to the target you want to debug.

1.4 Standard Stackdb Program Arguments

Programs built with Stackdb that use Stackdb's argument processing can supply standard, well-known arguments to quickly attach to targets and configure drivers. Stackdb's library is configured on a per-target basis, and it has no configuration language. Either your Stackdb programs use Stackdb's argument processing and augment it with any options they require; or they can directly configure it each time they open a target by populating Stackdb data structures directly. We strongly encourage you to use Stackdb's argument processing; it will simplify your life!

Here is a summary of the common Stackdb program arguments:

```
-a, --active-probing=FLAG,FLAG,...
                          A list of active probing flags to enable (disabled
                          by default) (thread_entry thread_exit memory other)
-E, --err-file=FILE       Log stderr (if avail) to FILE.
-I, --in-file=FILE        Deliver contents of FILE to target on stdin (if
                          avail).
```

```
-O, --out-file=FILE      Log stdout (if avail) to FILE.
-R, --debugfile-root-prefix=DIR
                         Set an alternate root prefix for debuginfo and
                         binfile resolution.
-d, --debug=LEVEL        Set/increase the debugging level
     (currently, levels run from 1 to 20 or so; higher number is
     more verbose output)
-F, --debugfile-load-opts=LOAD-OPTS
                         Add a set of debugfile load options.
-i, --target-id=ID       Specify a numeric ID for the target.
-l, --log-flags=FLAG,FLAG,...   Set the debugging flags.
    --personality=PERSONALITY   Forcibly set the target personality
                         (linux is the only one available).
    --personality-lib=PERSONALITY_LIB_FILENAME
                         Specify a shared library where the personality
                         specified by --personality should be loaded from.
-s, --soft-breakpoints   Force software breakpoints.
-t, --target-type=TYPENAME Forcibly set the target type
                         (ptrace,xen,gdb,os-process,php).
-w, --warn=LEVEL         Set/increase the warning level.
```

Typically, Stackdb drivers use *unique* option names – they don't share or overlap by design, if possible. Since Stackdb's argument processing is built using the GNU argp library, Stackdb will automatically select the driver matching the arguments you provide, if you do not specify a driver via the `-t` option. There are a few options that overlap; but as long as you pass a unique option first (for the driver you need to use), you don't need to specify `-t`. However, if the driver you want to use doesn't require any options at all, you'll need to specify it with `-t`.

If you need to augment your driver with a personality, use the `--personality` option. Currently, the only available personality is the `linux` personality, that can be used with drivers that provide access to OSes – such as the `xen` and `gdb` (with the `qemu` helper) drivers. If you develop your own personality outside the Stackdb source repository, you can specify a shared library filename where the personality can be found via `--personality-lib`.

1.4.1 Specifying Multiple Targets

Originally, Stackdb's argp argument processing was designed to be able to handle a single target passed on the command line, via the above arguments (and driver-specific arguments, discussed below). However, it is very useful at times to be able to specify multiple targets so that you can create stacks of targets. Many Stackdb programs support multiple targets (although a few are best suited to analyzing single targets only). Thus, for these kinds of programs that do support multiple targets, not only can you specify a "primary" target on the command

line; you can also use the `--base` and `--overlay` arguments to specify additional base targets, and additional overlay targets:

```
  --base=TARGET_OPTIONS  Specify an entire base target in a single
                         argument.  Any standard target option other than
                         --base and --overlay may be used.
  --overlay=OVERLAY_PREFIX:TARGET_OPTIONS
                         Specify an entire overlay target in a single
                         argument.  Your argument must be of the form
                         [<base_target_id>:]<thread_name_or_id>:TARGET_OPTIONS
```

So, for instance, you could invoke the `backtrace` tool like this:

```
$ backtrace --base '-t xen -m vm1 -i 10' --base '-t xen -m vm2 -i 20' \
    --overlay '10:bash:-t os-process' --overlay '20:php:-t os-process'
```

and thus obtain backtraces for each thread in the four targets.

1.4.2 Improving Debuginfo Loading Times

The dwdebug library's behavior can be controlled on a per-debuginfo-file basis. The `-F` option's value can be used to pass in an "rfilter" (a Stackdb regular expression filter) that can assign specific options to matching files. Typically, you'll be happier to globally set these options, rather than work with the powerful-yet-complex rfilters. Currently, the most useful option is `PARTIALSYM`. Stackdb's dwdebug library was designed to support very fast symbol and address lookups at runtime. Thus, it reads and indexes (by default) the entire debuginfo file. For a stock Ubuntu 3.8.x kernel, this can require over 1GB of RAM – the complete debuginfo for one of these Linux kernels is over 165MB when compressed – and indexing it unfortunately decompresses it. By enabling the `PARTIALSYM` option, the dwdebug library will only load symbols at the highest level of each compilation unit (essentially, each source file) – and it will expand them as necessary at runtime. This will result in slightly increased lookup times (the first time); but will result in significant memory savings. (We'll probably make this option the default someday, since expanded loading at runtime is fast anyway.)

```
$ ... -F PARTIALSYM ...
```

1.4.3 Debuginfo Search Path

If your Stackdb program is attaching to a target in userspace, or if the target program in your VM is running the same binaries as in the VM the Stackdb program is running in, your debuginfo will likely be installed in normal Linux locations in the filesystem (i.e., /usr/lib/debug). However, if you are attaching to an OS or process running in a VM, your VM may have different binaries than the host filesystem that the Stackdb program is running in. If this is true, you'll need to use the `-R` option to change the debuginfo search path prefix. You'll need

to create a directory that is a mirror of the relevant parts of the filesystem of the target program, that includes both the program/library binaries that make up the target program, and the corresponding debuginfo files for those binaries.

```
$ ... -R /path/to/debuginfo-filesystem-mirror ...
```

1.4.4 Active Probing

You can enable *active probing* within drivers that support it. Some drivers can get all the information they need to stay in control of the executing target program from the debugging API they attach to. For instance, the Ptrace driver can read all the information it needs about the target from the `ptrace` system call, and by reading the `/proc` filesystem. However, more complex drivers (i.e., the Xen or GDB/QEMU drivers that provide access to an OS) can do a better job of maintaining their model of the target if they *self-probe* – install probes inside the target so they are notified immediately when it changes in some debugger-relevant way. For instance, the generic Linux OS personality can install probes to detect when processes are created or are exiting (emulating ptrace(2), sort of), so that it can keep its list of threads up-to-date without scanning memory at each debug exception or user interrupt. Similarly, it can install probes on memory allocation-related system calls (mmap, mprotect, etc) to detect changes to process memory mappings.

If you construct stacks of targets, the overlay drivers may automatically enable active probing because they require it. However, you may have the option to do this, or to not do it. Much depends on your Stackdb program's analysis. If it makes frequent calls to target_load_all_threads(), you may be better off enabling active probing for `thread_entry` and `thread_exit` – unless your OS workload is *constantly* spawning processes. There is always a tradeoff between how often the self-probing active probes are hit, vs how long it takes to scan memory when required. Here's another scenario: the generic Linux OS personality will scan the Linux kernel's module list *on each debug exception*, looking for new modules so it can load their debuginfo. If you don't have much module loading/unloading churn, active probing is a big win, because these probes will never be hit! On the other hand, constantly rescanning the module list takes time, and is typically wasted.

Hopefully this guidance helps you get a feel for when and why active probing can be helpful; but you may be best-served by experimenting with your own Stackdb applications and your own workloads. Or, if performance overhead doesn't worry you, then this may not matter as much.

1.4.5 Standard Debugging Arguments

If you encounter a bug, the maintainers will likely ask you to produce a debug log. Typically, you'll start by enabling debug messages and optional warnings,

like

```
$ ... -d20 -w20 ...
```

Then you need to choose a set of log flags, which enable debug and warning messages in specific components and/or feature areas of Stackdb. Sometimes, the maintainers may recognize where a problem is likely to have originated, and can recommend a set of debug flags that will reduce the total logfile size while still providing enough information to find the bug. There are several log *areas*:

```
* LIB (L_*): covers components in the lib/ src dir;
* DEBUG (D_*): covers the dwdebug/ src dir;
* TARGET (T_*): the target/ src dir;
* PROBE (P_*): the probing (breakpoints, watchpoints) components in target/;
* XML (X_*): the xml components in xml/;
* USER (U_*): covers any user programs that wish to use Stackdb's
  debug/logging framework
```

The prefixes in parentheses are prepended to individual flag names, so that both a log area and a log flag (in that area) are specified to form a single flag name that can be included in the comma-separated list passed to the `-l` option. Here are the individual, per-area flags, with prefixes already prepended:

```
* LIB: L_CLMATCH, L_CLRANGE (the cl* data structures are used to
       index and search ranges and hierarchies of ranges -- for
       instance, a program's text segment is a series of nested
       ranges corresponding to function symbols), L_RFILTER,
       L_WAITPIPE, L_EVLOOP, L_MONITOR, L_REGCACHE (register cache
       functions, used by some drivers)
* DWDEBUG: D_DFILE (general debuginfo file issues), D_SYMBOL,
       D_SCOPE, D_LOC (symbol location issues), D_LOOKUP (symbol
       lookup), D_DWARF (general DWARF processing issues),
       D_DWARFATTR (DWARF attribute processing), D_DWARFOPS (DWARF
       operation processing -- DWARF includes a stack-machine
       expression language that can evaluate expressions to find
       locations of variables at runtime, for instance), D_DWARFSOPS
       (dwdebug attempts to pre-process trivial DWARF operation
       expressions when possible), D_CFA (CFA (aka CFI) --
       call frame information, used to unwind the call stack),
       D_OTHER, D_BFILE (generic binfile issues), D_ELF
       (ELF-specific binfile issues)
* TARGET: T_TARGET (generic target issues), T_SPACE (address
       spaces), T_REGION (memory regions), T_LOOKUP (symbol lookup),
       T_LOC (location decoding), T_OTHER, T_SYMBOL (symbols),
       T_MEMCACHE (memory caching support via mmap; some drivers use
       this generic support), T_UNW (unwinding), T_THREAD (threads),
       T_DISASM (disassembly), T_OS (OS personality issues),
       T_PROCESS (process personality issues), T_APPLICATION
```

```
       (application personality issues), T_LUP (ptrace driver), T_XV
       (xen driver), T_OSP (os-process overlay driver), T_GDB (gdb
       driver), T_PHP (php overlay driver)
* PROBE: P_PROBE (probes), P_PROBEPOINT (probes sit atop
       probepoints; probepoints implement low-level things like
       breakpoints), P_ACTION (single stepping is implemented as an
       action, for instance)
* XML: X_XML (general issues), X_RPC, X_SVC, X_PROXYREQ
* USER: U_ALL (basically, 0xffffffff -- Stackdb does not know a
       priori what the flag names should be for a user application
       -- right now the user can't discriminate specific flag values
       within the USER area)
```

1.5 Supported Platforms

1.5.1 Drivers

Stackdb's drivers allow you to debug several different kinds of targets. Some drivers are designed to stack atop other drivers, allowing you to debug a base target, and several levels of overlay targets. Here's a quick overview of the drivers we support; the targets they allow you to attach to; and limitations.

1.5.2 Linux Userspace Process (Ptrace) Driver

This driver is a `ptrace(2)` driver. It allows a Stackdb program to attach to a peer Linux process executing on the same machine as the Stackdb program. It uses the Linux/UNIX `ptrace(2)` system call to attach to and debug its target process. It supports multithreaded programs. It was built as a driver to enable testing, and to flesh out internal APIs; but it may be useful to you on its own. If you attach to a program that is running as your user ID, your Stackdb process does not need root privileges; otherwise, it will need them.

The Ptrace driver can either attach to existing processes; or it can launch a new process and attach to it. Here is a summary of the options this driver accepts:

```
    --args=LIST            A comma-separated argument list.
-b, --program=FILE         A program to launch as the target.
-e, --envvars=LIST         A comma-separated envvar list.
-p, --pid=PID              A target process to attach to.
```

You can also run your Stackdb program like this:

```
$ ./stackdb-program -t ptrace <myopt1> <myopt2> \
    -- ./program-to-debug arg1 arg2 ... argN
```

Stackdb will interpret all the arguments after the `--` as a vector to pass to execve, essentially. This is the most common and convenient way to launch and debug programs with the Ptrace driver. If you wish to debug an existing process, you can do something like this (if you have `pgrep` installed, which you almost certainly do):

```
$ ./stackdb-program -t ptrace -p `pgrep -n program-to-debug` \
    <myopt1> ... <myoptN>
```

1.5.3 Xen Driver

The Xen driver allows you to attach to a Xen VM. In this case, your Stackdb programs run in Domain 0, and they attach to other Xen VMs as necessary using Xen libraries (libxc, primarily). This driver provides different memory access mechanisms (available at runtime, depending on the ./configure options you supplied when building).

This driver may be supplemented by any OS personality. Its functionality is not especially useful without an OS personality; without a personality, the driver can only debug the currently executing thread, within its current memory context. However, if one is used, the Xen driver can support multiple threads, virtual and physical address spaces (and thus memory), and kernel and userspace thread contexts.

At the present time, the Xen driver only supports single-CPU VMs. This limitation is an artifact of our research project, not a fundamental limitation in Stackdb's design. Contact us if you're interested in or require such a configuration; if the need arises, we may consider building this support.

The Xen driver cannot launch VMs yet; it can only attach to them.

Here's a quick overview of the options available:

```
-C, --clear-libvmi-caches-each-rw
                          Clear libvmi caches on each memory-read/write.
    --hypervisor-ignores-userspace-exceptions
                          If your Xen hypervisor is not a Utah-patched
                          version, make sure to supply this flag!
-H, --no-clear-hw-debug-regs   Don't clear hardware debug registers at target attach.
-K, --kernel-filename=FILE Override xenstore kernel filepath for guest.
-m, --domain=DOMAIN        The Xen domain ID or name.
-M, --no-use-multiplexer   Do not spawn/attach to the Xen multiplexer server
-T, --dominfo-timeout=MICROSECONDS
                          If libxc gets a "NULL" dominfo status, the number of microseconds w
-V, --no-hvm-setcontext    Don't use HVM-specific libxc get/set context functions to accessv
-x, --xenlib-debug=LEVEL   Increase/set the XenAccess/OpenVMI debug level.
```

First, you must tell the driver which domain to attach to, via the `-m` option.

If you run your Xen VM in paravirtualized mode, then the Xen driver will be able to detect which kernel filename you are running and will attempt to find debuginfo for it in the standard location (or in the prefix you supplied with `-R`). However, if you are running an HVM, you must tell the driver the filename of the kernel your VM is running, via the `-K` option.

Note that the Xen driver's hardware breakpoints may not work for all Xen versions; if this happens to you, try to use the `-s` argument to force software breakpoints.

The Xen driver allows you to attach to multiple Xen VMs from Domain 0 (either within the same Stackdb program, or from within different Stackdb programs running at the same time). We need special code to do this because Xen only provides the ability for one Domain 0 program to listen for debug exceptions on other domains. Thus, the Xen driver provides a demultiplexing service that, when utilized, notifies all Stackdb programs attached to Xen VMs when there is a Xen debug exception. If you only need to attach to a single VM at one time, and don't want to incur any of the tiny, minimal overhead of the demultiplexing service, you can disable it via the `-M` option.

Another important option affects your ability to stack other drivers atop the Xen driver (i.e., to stack the os-process userspace-process driver atop the Xen driver, to debug processes in VMs from outside the VM). This option is `--hypervisor-ignores-userspace-exceptions`. It's not important if you're not stacking, though; you can attach to Xen VMs and debug the kernel without worrying about it. Here's a brief summary (the technical details are involved). Basically, the Xen hypervisor catches kernel-mode debug exceptions (int 3 and debug traps), and, if there's a debugger attached via the debugger VIRQ, it notifies the debugger and pauses the VM so the debugger can handle it. However, the hypervisor does not forward userspace debug exceptions to the debugger; it naturally expects that there's a userspace debugger running inside the VM that will handle the exceptions in cooperation with the guest kernel. Initially when developing stackdb, we patched the hypervisor to also forward userspace debug exceptions, but that doesn't easily let us support distro-packaged hypervisors. So we observed that since the hypervisor passes userspace exceptions back to the guest kernel (or the guest kernel gets them directly via HVM IDT), we could just install breakpoints on the guest kernel debug handlers, and emulate how they would handle a userspace exception (but we do the work if the userspace exception was a VMI-triggered exception) – and then immediately return from the handler in the kernel isntead of single-stepping the first instruction of the interrupt handler. So, VMI actually handles the userspace exception, but the kernel thinks it did. This works fine for HVM, but it is more complicated for PV domains, and it doesn't currently work. The hypervisor has to explicitly support this style of exceptions, and notice in which ring the exception occurred; this currently doesn't appear to be what the code does (and it's not what we observe at all). So this feature is not currently supported for PV domains, which means that if you want to use the os-process driver atop the Xen driver on a

PV domain, you'll need Utah's simple Xen patch, and then rebuild your Xen packages.

The other options mainly cover special cases and problems we've observed in Xen.

1.5.4 GDB/QEMU/KVM Driver

The GDB driver allows you to attach to any program that is coupled to a [GDB server stub] (and we use it to provide access to QEMU/KVM VMs) too. Many embedded systems, or more relevant to Stackdb, virtual machine hypervisors like Xen and QEMU/KVM, provide GDB server stubs that a GDB client can interact with to debug the embedded system, or the OS running inside a VM. Since each GDB server stub is different, and because Stackdb might require (or be able to leverage) additional platform information beyond what can be expressed in the GDB remote protocol), Stackdb's GDB driver allows the user to "plug in" *helper* modules.

Here are the primary options for the GDB driver:

```
    --gdb-host=HOST      The hostname the GDB stub is listening on
                         (default localhost).
    --gdb-port=PORT      The port the GDB stub is listening on (default 1234).
    --gdb-sockfile=FILE  The UNIX domain socket filename the GDB stub
                         is listening on.
    --gdb-udp            Use UDP instead of TCP (default TCP).
    --memcache-mmap-size=BYTES   Max size (bytes) of the mmap cache
                         (default 128MB).
-M, --main-filename=FILE Set main binary's filepath for target.
```

First, you must tell the driver the filename of the "main" program running inside the target, via the `-M` option. This is probably a kernel. The other important options to specify are `--gdb-host` and `--gdb-port`, if necessary.

1.5.4.1 QEMU GDB Helper

Stackdb supports part of the GDB remote debugging protocol, enough to provide debugging of QEMU/KVM VMs. To attach to QEMU/KVM VMs, the user must also specify the use of the QEMU *helper*. Currently, you must use our special QEMU support to use the QEMU helper. The QEMU helper relies on two additional non-GDB sources of information: 1) access to the VM's "physical" memory allocated by the QEMU process; and 2) access to the QEMU monitor port to load a few additional register values as needed.

Here are the QEMU helper's options:

```
    --qemu                 Enable QEMU GDB stub support
    --qemu-mem-path=FILE   Read/write QEMU's physical memory via this
```

```
                        filename (see QEMU's -mem-path option; also
                        preload libnunlink.so and set QEMU_MEMPATH_PREFIX
                        accordingly).
    --qemu-libvirt-domain=DOMAIN
                        Access QEMU QMP over libvirt proxy.
    --qemu-qmp-host=HOST   Attach to QEMU QMP on the given host
                        (default localhost).
    --qemu-qmp-port=PORT   Attach to QEMU QMP on the given port
                        (default 1235).
    --kvm               Enable KVM support.
```

1.5.4.2 QEMU/KVM Configuration

You must run your QEMU VM specifically to take advantage of Stackdb — although you do *not* have to modify QEMU itself! We have only tested our method with QEMU 2.0.x and 2.1.x; it may fail with other versions. Here's what you have to do.

First, follow instructions like those at http://www.linux-kvm.org/page/UsingLargePages to setup a hugetlbfs filesystem mounted at a location of your choosing; in the commands below, I assume you mount that filesystem at /hugetlbfs. Here's a quick set of commands that will help you get this going:

```
# Mount hugetlbfs at /hugetlbfs
$ sudo mkdir /hugetlbfs
$ sudo mount -t hugetlbfs hugetlbfs /hugetlbfs

# Reserve enough pages to it, for whatever your VMs
# will consume --- 512 will get you 1024M of 2M hugepages
$ echo 512 | sudo tee /proc/sys/vm/nr_hugepages

# Make sure the system was able to get you all 512 pages
# you asked for.  As the link above will tell you, it is
# best to mount hugetlbfs and reserve hugepages to it before
# your system's memory becomes fragmented.  One easy way to
# recover some memory is to kill Firefox!
$ cat /proc/meminfo | grep Huge
```

1.5.4.3 Manually Running QEMU/KVM

Once you have hugetlbfs setup, run your QEMU VM:

```
$ sudo QEMU_MEMPATH_PREFIX=/hugetlbfs/qemu \
       LD_PRELOAD=/path/to/stackdb-build-dir/target/.libs/libqemuhacks.so.0.0.0 \
  qemu-system-x86_64 -cpu host -m 512 -enable-kvm \
      -kernel /boot/vmlinuz-2.6.18-308.el5 -initrd /boot/automfs-2.6.18-308.el5.img \
      -append console=ttyS0 -nographic \
```

```
-gdb tcp::1234 -qmp tcp:127.0.0.1:1235,server,nowait -mem-path /hugetlbfs
```

QEMU_MEMPATH_PREFIX is an environment variable that is looked for by libqemuhacks.so.0.0.0, which is itself LD_PRELOADed before the qemu-system-x86_64 program runs. libqemuhacks.so.0.0.0 simply ensures that the file that QEMU creates in /hugetlbfs (or whatever you set `-mem-path` to above in your QEMU command) as a backing store for your VM's physical RAM is 1) mmap'd with MAP_SHARED instead of MAP_PRIVATE so that other processes can mmap it too; and 2) that the file is not immediately unlink()'d after QEMU opens it. Thus, it interposes on mmap and unlink, and ensures that any filename passed to those system calls that starts with the value in QEMU_MEMPATH_PREFIX is 1) mmap'd with MAP_SHARED, and 2) is not actually unlinked, so it stays present in the filesystem. libqemuhacks.so does try to remove those files when the QEMU process terminates, but if the process is sent a SIGKILL, it cannot catch the dying/exiting process — and in this case, you'll need to manually remove those files. libqemuhacks.so supports different strategies to try to remove these files. First, it interposes on the signals QEMU monitors to terminate itself (the same code path is followed if the QEMU process is signaled externally, or the code inside the machine shuts it down); this is the default and best option. It can also install an atexit() handler to try to remove these files, but that doesn't do anything for QEMU 2.x, because QEMU self-terminates by signaling its process. Finally, of course, you must also set LD_PRELOAD to preload libqemuhacks.so.0.0.0.

Then, once your VM has started, a file like `/hugetlbfs/qemu_back_mem.pc.ram.*` should appear; that is the file that Stackdb will try to mmap to obtain direct access to the VM's "physical" memory. When you run a Stackdb program to attach to QEMU, your command will look like

```
$ sudo gdb --args dumptarget -t gdb --qemu \
      --gdb-port 1234 --qemu-qmp-port 1235 \
      --qemu-mem-path /hugetlbfs/qemu_back_mem.pc.ram.AMytA1 \
      --kvm -M /tftpboot/vmlinux-syms-2.6.18-308.el5 \
      -s sys_open
```

(We do not depend on hugetlbfs nor hugepages specifically; we don't care about that; however, the –mem-path option only works on a hugetlbfs mountpoint. All we care about is that QEMU leaves us with a file we can mmap to get physical memory access — and this was the only way to get access without hacking QEMU!)

1.5.4.4 Using Libvirt to Run QEMU/KVM

First, make sure you've read the section above on running QEMU/KVM manually; this will explain the necessary environment variables and command-line options. We need to customize the libvirt VM config file to allow Stackdb to attach to it.

One of my config files (`/etc/libvirt/qemu/vm1.xml`) looks like this:

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
  <name>vm1</name>
  <uuid>29bfac01-b24d-e4ab-e741-f33f7e880d9d</uuid>
  <memory unit='KiB'>524288</memory>
  <currentMemory unit='KiB'>524288</currentMemory>

  <memoryBacking>
    <hugepages/>
  </memoryBacking>

  <vcpu placement='static'>1</vcpu>
  <os>
    <type arch='x86_64' machine='pc-i440fx-2.0'>hvm</type>
    <kernel>/tftpboot/roots/centos5.5-x86_64/boot/vmlinuz-2.6.18-308.el5</kernel>
    <initrd>/tftpboot/roots/centos5.5-x86_64/boot/initrd-2.6.18-308-full.el5.img</initrd>
    <cmdline>"console=tty0 console=ttyS0,115200n8"</cmdline>
    <boot dev='hd'/>
  </os>
  <features>
    <acpi/>
    <pae/>
  </features>
  <clock offset='utc'/>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>restart</on_crash>
  <devices>
    <emulator>/usr/bin/qemu-system-x86_64</emulator>
    <controller type='usb' index='0'>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x01' function='0x2'/>
    </controller>
    <controller type='pci' index='0' model='pci-root'/>
    <serial type='pty'>
      <target port='0'/>
    </serial>
    <console type='pty'>
      <target type='serial' port='0'/>
    </console>
    <input type='mouse' bus='ps2'/>
    <input type='keyboard' bus='ps2'/>
    <graphics type='vnc' port='-1' autoport='yes' keymap='en-us'/>
    <video>
      <model type='cirrus' vram='9216' heads='1'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x03' function='0x0'/>
    </video>
    <memballoon model='virtio'>
```

```
      <address type='pci' domain='0x0000' bus='0x00' slot='0x02' function='0x0'/>
    </memballoon>
  </devices>

  <qemu:commandline>
    <qemu:arg value='-gdb'/>
    <qemu:arg value='tcp:127.0.0.1:1234,nowait,nodelay,server'/>
    <qemu:env name='QEMU_MEMPATH_PREFIX' value='/hugetlbfs/'/>
    <qemu:env name='LD_PRELOAD' value='/home/johnsond/g/a3/vmi.master.obj/target/.libs/libqe
  </qemu:commandline>

</domain>
```

The first important bit is

```
  <memoryBacking>
    <hugepages/>
  </memoryBacking>
```

This will execute QEMU with hugepage support. Current libvirt distributions automatically detect your `hugetlbfs` mountpoint, and will place a directory in it for QEMU VMs to use, like `/hugetlbfs/libvirt/qemu`. This directory must be readable/writeable by the `qemu` user on my install; yours is probably similar. If this doesn't happen automatically, you can manually tell libvirt where the hugetlbfs mountpoint is, by changing the following line in `/etc/libvirt/qemu.conf`:

```
hugetlbfs_mount = "/hugetlbfs"
```

The second important bit is

```
  <qemu:commandline>
    <qemu:arg value='-gdb'/>
    <qemu:arg value='tcp:127.0.0.1:1234,nowait,nodelay,server'/>
    <qemu:env name='QEMU_MEMPATH_PREFIX' value='/hugetlbfs/'/>
    <qemu:env name='LD_PRELOAD' value='/home/johnsond/g/a3/vmi.master.obj/target/.libs/libqe
  </qemu:commandline>
```

This tells the QEMU process to start up its GDB stub, and listen on the host/port indicated; and to set the two environment variables we need. Change the value for LD_PRELOAD to the location where libqemuhacks.so.0.0.0 is on your system.

1.5.4.5 Using Eucalyptus to Run QEMU/KVM

Eucalyptus uses Libvirt to manage VMs. When Eucalyptus creates a VM (i.e., if you use the `euca-install-image` command, or the Eucalyptus web console), Eucalyptus will take the input, then map a Libvirt XML file template, and

finally generate an XML file that could be taken as an input by Libvirt APIs. We need to modify the Libvirt XML file template as done in the previous section.

The template file is probably /etc/eucalyptus/libvirt.xsl . This is a template to match all the Eucalyptus options, so you would see many "if" clauses in it. You would find below three lines, which checks whether the hypervisor type is Xen or KVM.

```
<domain>
  <xsl:attribute name="type">
    <xsl:value-of select="/instance/hypervisor/@type"/>
  </xsl:attribute>
```

Since Xen on Eucalyptus isn't well-supported, and since we only want it to work for KVM here, you could safely comment out those three lines, and add the extra lines for KVM like in last section.

Then, add several lines right after the lines being commented out:

```
<domain type="kvm" xmlns:qemu="http://libvirt.org/schemas/domain/qemu/1.0">
  <memoryBacking>
    <hugepages/>
  </memoryBacking>
  <qemu:commandline>
    <qemu:arg value='-gdb'/>
    <qemu:arg value='tcp:127.0.0.1:1234,nowait,nodelay,server'/>
    <qemu:env name='QEMU_MEMPATH_PREFIX' value='/hugetlbfs/'/>
    <qemu:env name='LD_PRELOAD' value='/home/mind/Downloads/vmi/stackDB/vmi.obj/target/.libs
  </qemu:commandline>
```

Then you can create Eucalyptus VMs as before, using "euca-run-instances" or web console; the generated memory file will be in `/hugetlbfs/libvirt/qemu`. Note that since the tcp port is specified as `1234` in the XML template, so when you simply use the same way to create Eucalyptus VM again, it would fail since the specific tcp port has been taken, so remember to change the file to some other port number if you want multiple VMs running which all could use StackDB. (There might be a way to modify the Eucalyptus source code or the template itself in order to automatically change the port number, but we haven't needed that yet.)

1.5.4.6 Using OpenStack to Run QEMU/KVM

[TBD.]

1.5.5 OS Process Driver

There is no special configuration necessary to use this driver — but you must ensure that the underlying base driver is configured with an OS personality! This currently happens automatically, and is assumed.

(*However*, if you're trying to use the OS Process driver atop a Xen PV domain, you'll need to read the Xen driver section, particularly the part about the `--hypervisor-ignores-userspace-exceptions` option!

1.5.6 PHP Driver

There is no special configuration necessary to use this driver. However, it does not support many Stackdb features, and only supports a tiny subset of PHP. You probably shouldn't use this driver except for global or member function breakpoints, unless you're willing to hack it!

1.5.7 Personalities

1.5.7.1 "Generic" Linux OS Personality

The only personality currently available is the Linux OS personality, which provides the OS personality interface. We have tested it on Linux kernels 2.6.18, 2.6.32, 3.2.x, and 3.8.x. It may well work on kernels in between, or it may not. The reason it may not work for every kernel is because it relies on the presence of specific symbols and data types to obtain some of its information to provide a full model of the running kernel to Stackdb. When these symbols change, Stackdb must account for them. Fortunately, Stackdb's symbol set is quite small and unlikely to change.

1.6 Supported Configurations

Remember, Stackdb is designed to allow you to stack targets atop each other to debug an entire software stack. You can use each of the base drivers (xen, gdb, ptrace) to attach to individual targets; and you can stack the overlay drivers atop them. The OS-Process driver can sit atop any base driver that provides an OS personality (the Xen and GDB/KVM/QEMU drivers do, when combined with the Linux OS personality); and the PHP driver can sit atop either the OS-Process or Ptrace drivers, since they provide access to processes, and PHP runs in a process. Here are two diagrams that may help you visualize how stacked driver configurations work: stack of Xen, OS-Process, and PHP drivers; stack of Ptrace and PHP drivers

1.7 Working with Debuginfo

In order for Stackdb to be a useful debugger and memory forensics tool, it must provide source-level debugging. Without source- and type-knowledge of the target program you are analyzing, it is difficult and time-consuming to figure out which memory to read and how to interpret it.

If you've ever used GDB to debug a crashing program that you (or someone else) wrote, you've probably used GDB's `bt` command to obtain a backtrace. If you hadn't installed the debugging symbols for the binaries and libraries used in the program, you would at best see a backtrace with function addresses and names:

```
$ gdb --args bash
GNU gdb (Ubuntu 7.7-0ubuntu3.1) 7.7
Copyright (C) 2014 Free Software Foundation, Inc.
...
Reading symbols from bash...(no debugging symbols found)...done.
(gdb) b make_child
Breakpoint 1 at 0x446450
(gdb) r
Starting program: /bin/bash

Breakpoint 1, 0x0000000000446450 in make_child ()
(gdb) bt
#0  0x0000000000446450 in make_child ()
#1  0x000000000044e507 in command_substitute ()
#2  0x000000000045167e in ?? ()
#3  0x00000000004548ea in ?? ()
#4  0x0000000000455777 in ?? ()
#5  0x0000000000455fec in ?? ()
#6  0x00000000004560ac in expand_word_unsplit ()
#7  0x00000000004360cc in execute_command_internal ()
#8  0x000000000043784e in execute_command ()
#9  0x0000000000435fc7 in execute_command_internal ()
#10 0x0000000000478350 in parse_and_execute ()
#11 0x0000000000477be3 in ?? ()
#12 0x0000000000477de7 in maybe_execute_file ()
#13 0x000000000041fa20 in main ()
(gdb)
```

However, you would not have seen a backtrace with function argument values, unless you had installed debug symbols. Moreover, you would not have been able to examine each frame's local variable values without debug symbols. Since we're running GDB on an Ubuntu system, we can simply follow the instructions for working with Ubuntu Debug Symbol Packages:

```
echo "deb http://ddebs.ubuntu.com $(lsb_release -cs) main restricted universe multiverse" \
```

```
    | sudo tee -a /etc/apt/sources.list.d/ddebs.list
echo "deb http://ddebs.ubuntu.com $(lsb_release -cs)-updates main restricted universe multiv
    | sudo tee -a /etc/apt/sources.list.d/ddebs.list
echo "deb http://ddebs.ubuntu.com $(lsb_release -cs)-proposed main restricted universe multi
    | sudo tee -a /etc/apt/sources.list.d/ddebs.list
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 428D7C01
sudo apt-get update
```

Then we can install `bash`'s debug symbols like this:

```
sudo apt-get install bash-dbgsym
```

Then our GDB backtrace becomes much more useful:

```
$ gdb --args bash
GNU gdb (Ubuntu 7.7-0ubuntu3.1) 7.7
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
...
Reading symbols from bash...Reading symbols from /usr/lib/debug//bin/bash...done.
done.
(gdb) b make_child
Breakpoint 1 at 0x446450: file .././jobs.c, line 1717.
(gdb) r
Starting program: /bin/bash

Breakpoint 1, make_child (command=command@entry=0x0, async_p=0) at .././jobs.c:1717
1717    .././jobs.c: No such file or directory.
(gdb) bt
#0  make_child (command=command@entry=0x0, async_p=0) at .././jobs.c:1717
#1  0x000000000044e507 in command_substitute (string=string@entry=0x70db88 "groups", quoted=
#2  0x000000000045167e in param_expand (string=string@entry=0x70da48 " $(groups) ", sindex=s
    expanded_something=expanded_something@entry=0x0, contains_dollar_at=contains_dollar_at@e
    quoted_dollar_at_p=quoted_dollar_at_p@entry=0x7fffffffde00, had_quoted_null_p=had_quoted
    at .././subst.c:7952
#3  0x00000000004548ea in expand_word_internal (word=word@entry=0x70dd28, quoted=quoted@entr
    contains_dollar_at=contains_dollar_at@entry=0x7fffffffdf28, expanded_something=expanded_
#4  0x0000000000455777 in expand_word_internal (word=word@entry=0x70c468, quoted=<optimized
    contains_dollar_at=contains_dollar_at@entry=0x0, expanded_something=expanded_something@e
#5  0x0000000000455fec in call_expand_word_internal (w=0x70c468, q=<optimized out>, i=i@entr
#6  0x00000000004560ac in expand_word_unsplit (word=<optimized out>, quoted=<optimized out>)
    #7  0x00000000004360cc in execute_case_command (case_command=<optimized out>) at .././ex
#8  execute_command_internal (command=0x70cdc8, asynchronous=7392648, pipe_in=-1, pipe_out=7
#9  0x000000000043784e in execute_command (command=0x70cdc8) at .././execute_cmd.c:390
#10 0x0000000000435fc7 in execute_if_command (if_command=<optimized out>) at .././execute_cm
#11 execute_command_internal (command=0x70ce48, asynchronous=7392776, pipe_in=-1, pipe_out=5
#12 0x0000000000478350 in parse_and_execute (string=<optimized out>, from_file=from_file@ent
    at ../../../builtins/evalstring.c:367
```

```
#13 0x0000000000477be3 in _evalfile (filename=filename@entry=0x6fe148 "/etc/bash.bashrc", fl
#14 0x0000000000477de7 in maybe_execute_file (fname=fname@entry=0x4b50d7 "/etc/bash.bashrc",
    at ../.././builtins/evalfile.c:320
#15 0x000000000041fa20 in run_startup_files () at .././shell.c:1134
#16 main (argc=1, argv=0x7fffffffe6e8, env=0x7fffffffe6f8) at .././shell.c:655
(gdb)
```

We can also load and examine variable values trivially:

```
(gdb) p environ
$1 = (char **) 0x7fffffffe6f8
(gdb) p environ[0]
$2 = 0x7fffffffe8fd "XDG_SESSION_ID=1"
(gdb) p environ[1]
$3 = 0x7fffffffe90e "SHELL=/bin/bash"
(gdb) p environ[2]
$4 = 0x7fffffffe91e "TERM=xterm-color"
(gdb)
```

We can also examine type information:

```
(gdb) ptype make_child
type = int (char *, int)
(gdb)
```

Of course, GDB can do many other things as well, and so can Stackdb. But the point here is that a debugger is much more powerful when you make debug symbols available to it.

1.7.1 Installing Debuginfo Packages on Ubuntu Systems

Read and refer to Ubuntu Debug Symbol Packages.

1.7.2 Installing Debuginfo Packages on Fedora/CentOS Systems

Read and refer to Fedora Debug Symbol Packages.

1.8 Stackdb Tools

Stackdb provides some utility programs, as well as some "real" programs that perform useful analyses of a target. We discuss these in the following sections.

1.8.1 dumpdebuginfo

`dumpdebuginfo` is the only `dwdebug`-specific tool Stackdb provides. It does not provide access to targets; but instead reads, parses, indexes, and displays debuginfo files. It displays types and symbols in the hierarchies in which they are present in the source program.

```
Usage: dumpdebuginfo [options] filename [symbols or addresses to look up]

  -d    Increase debug level
  -dN   Set debug level to N
  -w    Increase optional warning level
  -wN   Set optional warning level to N
  -l FLAG,FLAG,...       A comma-separated list of debug area flags
  -D        Increase the level of detail shown
  -M    Increase the level of metadata shown (i.e., location info)
  -F    Specify debuginfo load options; an rfilter
  -T    Disable display of global types
  -G    Disable display of global variables/functions
  -S    Disable display of symbol tables
  -R    Disable display of ranges
  -E    Disable display of ELF symbol tables
  -N    Don't waste time freeing data structures on exit
  -I N:ADDR,N:ADDR,...
    Tell the ELF library that section N was loaded at ADDR
  -i ADDR   Specify a base address
```

Here's a minimal "Hello, world!" program:

```
#include <stdio.h>

int main(int argc,char **argv) {
    printf("Hello, world!\n");
    return 0;
}
```

Here's an example of running `dumpdebuginfo` on this minimal program (slightly tweaked to break up the very long lines normally present). The main things to take away from this example are the structure of the debuginfo, that mirrors the structure of the code; the fact that many symbols have locations, and some symbols are only available in certain text segments (and maybe be in a different location depending on which text segment the processor is executing!); the extensive type information. Basically, if you need to find a symbol, or to see if a symbol is actually available at a given text address within a function, dumpdebuginfo can be very helpful. Note that in addition to passing the debuginfo filename, you can also specify symbols and/or addresses to look up; in that case, only the search results will be printed, not the entire debuginfo file

dump. When you search for a symbol, you may search for things like `a.b.c`, where a is a struct; b is a member of typeof(a); and c is a member of typeof(c). Moreover, you may use the '.' operator to indirect across pointers. Thus, if 'b' is actually a pointer to another struct type, which contains 'c' as a member, this search will still return a chain of symbols (an lsymbol).

```
$ dumpdebuginfo -DM hello_world

debugfile(/home/johnsond/hello_world):
    flags:  0x0
    refcnt:   1
  types: (12)
    type:  (4 B) (encoding=5)
    type:  (8 B) (encoding=7)
    type:  (1 B) (encoding=6)
    type:  (8 B) (encoding=5)
    type:  (4 B) (encoding=7)
    type:  (2 B) (encoding=5)
    type:  (8 B) (encoding=7)
    type: typedef  (8 B) (encoding=7) size_t (line=213)
    type:  (1 B) (encoding=6)
    type:  (2 B) (encoding=7)
    type:  (1 B) (encoding=8)
    type: void
  shared_types: (0)
  globals: (3)
    func: void __libc_csu_fini () (external,prototyped,frame_base=R7+8,line=95,) @@ 0x40062
              scope(__libc_csu_fini @@ 0x400620) RANGES([0x400620,0x400622]) { }
    func: int main (int argc (line=3,) @@ FB-20, char** argv (line=3,) @@ FB-32)
              (external,prototyped,frame_base=LIST([0x40056c,0x40056d->R7+8],[0x40056d,0x400
               [0x400570,0x40058b->R6+16],[0x40058b,0x40058c->R7+8]),line=3,) @@ 0x40056c
              scope(main @@ 0x40056c) RANGES([0x40056c,0x40058c]) { }
    func: void __libc_csu_init (int argc (line=67,)
                                  @@ LIST([0x400590,0x4005cd->R5],[0x4005cd,0x400614->R15],
                                          [0x400614,0x400619->RUNTIMEDATA(4,f31559f)]),
                                char** argv (line=67,)
                                  @@ LIST([0x400590,0x4005cd->R4],[0x4005cd,0x40060f->R14],
                                          [0x40060f,0x400619->RUNTIMEDATA(4,f31549f)]),
                                char** envp (line=67,)
                                  @@ LIST([0x400590,0x4005cd->R1],[0x4005cd,0x40060a->R13],
                                          [0x40060a,0x400619->RUNTIMEDATA(4,f31519f)]))
              (external,prototyped,frame_base=LIST([0x400590,0x4005bb->R7+8],
               [0x4005bb,0x400618->R7+64],[0x400618,0x400619->R7+8]),line=67,) @@ 0x400590
              scope(__libc_csu_init @@ 0x400590) RANGES([0x400590,0x400619]) { }
  root srcfiles: (2)
    root: hello_world.c (compdirname=/home/johnsond,producer=GNU C 4.7.3,language=C89 (1)) +
```

```
  scope(hello_world.c @@ 0x40056c) RANGES([0x40056c,0x40058c]) {
    symbols: {
      type:  (4 B) (encoding=5)
      type:  (8 B) (encoding=7)
      type:  (1 B) (encoding=6)
      type:  (8 B) (encoding=5)
      func: int main (int argc (line=3,) @@ FB-20,
                      char** argv (line=3,) @@ FB-32)
                 (external,prototyped,frame_base=LIST([0x40056c,0x40056d->R7+8],
                  [0x40056d,0x400570->R7+16],[0x400570,0x40058b->R6+16],
                  [0x40058b,0x40058c->R7+8]),line=3,) @@ 0x40056c
                 scope(main @@ 0x40056c) RANGES([0x40056c,0x40058c]) { }
      type:  (4 B) (encoding=7)
      type:  (2 B) (encoding=5)
      type:  (8 B) (encoding=7)
      type:  (1 B) (encoding=6)
      type:  (2 B) (encoding=7)
      type:  (1 B) (encoding=8)
      type: void
  }
  subscopes: {
    scope(main @@ 0x40056c) RANGES([0x40056c,0x40058c]) {
      symbols: {
          var: char** argv (line=3,) @@ FB-32
          var: int argc (line=3,) @@ FB-20
      }
    }
  }
}
}

root: elf-init.c (compdirname=/var/tmp/portage/sys-libs/glibc-2.19-r1/work/glibc-2.19/cs
                  producer=GNU C 4.7.3,language=C89 (1)) {
  scope(elf-init.c @@ 0x400590) RANGES([0x400590,0x400622]) {
    symbols: {
      type:  (4 B) (encoding=5)
      type:  (1 B) (encoding=6)
      type:  (8 B) (encoding=7)
      func: void __libc_csu_init (int argc (line=67,)
                                    @@ LIST([0x400590,0x4005cd->R5],[0x4005cd,0x400614->
                                            [0x400614,0x400619->RUNTIMEDATA(4,f31559f)])
                                  char** argv (line=67,)
                                    @@ LIST([0x400590,0x4005cd->R4],[0x4005cd,0x40060f->
                                            [0x40060f,0x400619->RUNTIMEDATA(4,f31549f)])
                  char** envp (line=67,)
                 @@ LIST([0x400590,0x4005cd->R1],[0x4005cd,0x40060a->R13],
```

```
                          [0x40060a,0x400619->RUNTIMEDATA(4,f31519f)]))
                    (external,prototyped,frame_base=LIST([0x400590,0x4005bb->R7+8],
                     [0x4005bb,0x400618->R7+64],[0x400618,0x400619->R7+8]),line=67,) @@ 0x40
                    scope(__libc_csu_init @@ 0x400590) RANGES([0x400590,0x400619]) { }
        func: _init () (external,prototyped,decl,line=55,)
        type:  (8 B) (encoding=5)
        var: ()*[<UNK>] __init_array_end (external,isdecl,line=46,)
        var: ()*[<UNK>] __init_array_start (external,isdecl,line=44,)
        type: typedef  (8 B) (encoding=7) size_t (line=213)
        func: void __libc_csu_fini () (external,prototyped,frame_base=R7+8,line=95,) @@ 0x
                  scope(__libc_csu_fini @@ 0x400620) RANGES([0x400620,0x400622]) { }
        type: void
    }
    subscopes: {
      scope(__libc_csu_init @@ 0x400590) RANGES([0x400590,0x400619]) {
        symbols: {
            var: const size_t size (line=86,)
            var: char** argv (line=67,)
                   @@ LIST([0x400590,0x4005cd->R4],[0x4005cd,0x40060f->R14],
                           [0x40060f,0x400619->RUNTIMEDATA(4,f31549f)])
            var: char** envp (line=67,)
                   @@ LIST([0x400590,0x4005cd->R1],[0x4005cd,0x40060a->R13],
                           [0x40060a,0x400619->RUNTIMEDATA(4,f31519f)])
            var: int argc (line=67,)
                   @@ LIST([0x400590,0x4005cd->R5],[0x4005cd,0x400614->R15],
                           [0x400614,0x400619->RUNTIMEDATA(4,f31559f)])
        }
        subscopes: {
          scope() RANGES([0x4005bb,0x4005c9][0x4005ce,0x4005f6]) {
            symbols: {
              var: size_t i (line=87,)
                     @@ LIST([0x4005ce,0x4005dc->RUNTIMEDATA(2,309f)],[0x4005f1,0x4005f6->
            }
          }
        }
      }
      scope(__libc_csu_fini @@ 0x400620) RANGES([0x400620,0x400622]) { }
      scope((null)) { }
    }
  }
  }


multi-use srcfile symtabs: (0)
ranges:
  0x40056c,0x40058c: hello_world.c
```

```
  0x400590,0x400622: elf-init.c
binfile root: (tab=47,anon=0,dup=0,subscopes=0)
  root: /home/johnsond/hello_world (compdirname=(null),producer=(null),language=(null) (0)
    scope(/home/johnsond/hello_world) {
      symbols: {
        var:  .bss (external,) @@ 0x601040 (8 B)
        var:  _end (external,) @@ 0x601048 (0 B)
        func: .fini () (external,prototyped,) @@ 0x400624 (9 B)
        func: __gmon_start__@plt () (external,prototyped,) @@ 0x400470 (16 B)
        var:  .rela.plt (external,) @@ 0x4003d8 (72 B)
        var:  .got (external,) @@ 0x600ff8 (8 B)
        func: _fini () (external,prototyped,) @@ 0x400624 (9 B)
        var:  _GLOBAL_OFFSET_TABLE_ @@ 0x601000 (48 B)
        var:  __TMC_END__ (external,) @@ 0x601040 (8 B)
        func: .plt () (external,prototyped,) @@ 0x400440 (64 B)
        var:  .hash (external,) @@ 0x4002b0 (36 B)
        func: .init () (external,prototyped,) @@ 0x400420 (26 B)
        func: .text () (external,prototyped,) @@ 0x400480 (420 B)
        func: _init () (external,prototyped,) @@ 0x400420 (26 B)
        var:  .init_array (external,) @@ 0x600e00 (8 B)
        var:  .dynstr (external,) @@ 0x400358 (61 B)
        func: puts@plt () (external,prototyped,) @@ 0x400450 (16 B)
        var:  .got.plt (external,) @@ 0x601000 (48 B)
        var:  __init_array_end @@ 0x600e08 (8 B)
        var:  .eh_frame (external,) @@ 0x400678 (212 B)
        var:  _IO_stdin_used (external,) @@ 0x400630 (4 B)
        var:  .dynamic (external,) @@ 0x600e18 (480 B)
        func: __libc_csu_fini () (external,prototyped,) @@ 0x400620 (2 B)
        var:  .jcr (external,) @@ 0x600e10 (8 B)
        var:  _DYNAMIC @@ 0x600e18 (480 B)
        var:  .interp (external,) @@ 0x400270 (28 B)
        func: __libc_csu_init () (external,prototyped,) @@ 0x400590 (137 B)
        var:  .rodata (external,) @@ 0x400630 (18 B)
        func: __libc_start_main@plt () (external,prototyped,) @@ 0x400460 (16 B)
        var:  __dso_handle (external,) @@ 0x601038 (8 B)
        var:  __init_array_start @@ 0x600e00 (8 B)
        var:  data_start (external,) @@ 0x601030 (8 B)
        var:  __bss_start (external,) @@ 0x601040 (8 B)
        var:  .note.ABI-tag (external,) @@ 0x40028c (32 B)
        var:  .eh_frame_hdr (external,) @@ 0x400644 (52 B)
        var:  .fini_array (external,) @@ 0x600e08 (8 B)
        var:  .rela.dyn (external,) @@ 0x4003c0 (24 B)
        func: _start () (external,prototyped,) @@ 0x400480 (236 B)
        var:  .gnu.version_r (external,) @@ 0x4003a0 (32 B)
        var:  __data_start (external,) @@ 0x601030 (8 B)
        var:  _edata (external,) @@ 0x601040 (8 B)
```

```
        var:  .gnu.hash (external,) @@ 0x4002d8 (28 B)
        func: main () (external,prototyped,) @@ 0x40056c (32 B)
        var:  .dynsym (external,) @@ 0x4002f8 (96 B)
        var:  .gnu.version (external,) @@ 0x400396 (8 B)
        func: _header@plt () (external,prototyped,) @@ 0x400440 (16 B)
        var:  .data (external,) @@ 0x601030 (16 B)
    }
  }
  }

binfile root ranges:
  0x400270,0x40028c: .interp
  0x40028c,0x4002ac: .note.ABI-tag
  0x4002b0,0x4002d4: .hash
  0x4002d8,0x4002f4: .gnu.hash
  0x4002f8,0x400358: .dynsym
  0x400358,0x400395: .dynstr
  0x400396,0x40039e: .gnu.version
  0x4003a0,0x4003c0: .gnu.version_r
  0x4003c0,0x4003d8: .rela.dyn
  0x4003d8,0x400420: .rela.plt
  0x400420,0x40043a: _init
  0x400420,0x40043a: .init
  0x400440,0x400480: .plt
  0x400440,0x400450: _header@plt
  0x400450,0x400460: puts@plt
  0x400460,0x400470: __libc_start_main@plt
  0x400470,0x400480: __gmon_start__@plt
  0x400480,0x40056c: _start
  0x400480,0x400624: .text
  0x40056c,0x40058c: main
  0x400590,0x400619: __libc_csu_init
  0x400620,0x400622: __libc_csu_fini
  0x400624,0x40062d: _fini
  0x400624,0x40062d: .fini
  0x400630,0x400634: _IO_stdin_used
  0x400630,0x400642: .rodata
  0x400644,0x400678: .eh_frame_hdr
  0x400678,0x40074c: .eh_frame
  0x600e00,0x600e08: __init_array_start
  0x600e00,0x600e08: .init_array
  0x600e08,0x600e10: __init_array_end
  0x600e08,0x600e10: .fini_array
  0x600e10,0x600e18: .jcr
  0x600e18,0x600ff8: _DYNAMIC
  0x600e18,0x600ff8: .dynamic
```

```
    0x600ff8,0x601000: .got
    0x601000,0x601030: _GLOBAL_OFFSET_TABLE_
    0x601000,0x601030: .got.plt
    0x601030,0x601038: data_start
    0x601030,0x601038: __data_start
    0x601030,0x601040: .data
    0x601038,0x601040: __dso_handle
    0x601040,0x601048: _edata
    0x601040,0x601048: __bss_start
    0x601040,0x601048: __TMC_END__
    0x601040,0x601048: .bss
    0x601048,0x601048: _end
  binfile_pointing root: (tab=0,anon=0,dup=0)
```

1.8.2 dumptarget

The `dumptarget` program is the original "test" program we evolved as we developed Stackdb. We don't recommend using it, but it can be a useful tool to place probes on functions, addresses, sourcefile/lines, and variables — using the "old" `target_monitor()` style (a single, blocking event loop — probetargets, for instance, uses `target_monitor_evloop()` to be able to monitor multiple targets at once). Basic usage would look like

```
$ dumptarget -t gdb --qemu --kvm --gdb-host 127.0.0.1 --gdb-port 1234 \
    --qemu-libvirt-domain vm1 --qemu-mem-path /hugetlbfs/libvirt/qemu/qemu_back_mem.pc.ram.*
    -M /tftpboot/roots/centos5.5-x86_64/boot/vmlinux-syms-2.6.18-308.el5 \
    --overlay 'bash:-t os-process -s -R /tftpboot/roots/centos5.5-x86_64'
    sys_open make_child
```

to place probes on `sys_open` in the Linux kernel in the base target, and on `make_child` in the bash process the overlay target is attached to.

1.8.3 probetargets

The `probetargets` program is a modern, less-complicated version of `dumptarget`, but it only accepts symbols to place probes on. However, you can also prefix each symbol with the ID of a target. So, you could run the command above like this:

```
$ probetargets \
    --base '-t gdb -i 10 --qemu --kvm --gdb-host 127.0.0.1 --gdb-port 1234 \
            --qemu-libvirt-domain vm1 \
            --qemu-mem-path /hugetlbfs/libvirt/qemu/qemu_back_mem.pc.ram.* \
            -M /tftpboot/roots/centos5.5-x86_64/boot/vmlinux-syms-2.6.18-308.el5' \
    --overlay '10:bash:-t os-process -i 20 -s -R /tftpboot/roots/centos5.5-x86_64'
    10:sys_open 20:make_child
```

because you want to place the `sys_open` probe on target id 10, and the `make_child` probe on target id 20.

1.8.4 dumpthreads

`dumpthreads` can dump (detailed) information about each thread in a target, and it can loop and dump threads at a given interval.

1.8.5 backtrace

`backtrace` can dump backtraces for target threads.

1.8.6 spf

`spf` is a generic, multi-target, multi-stack debugger with a very simple configuration file interface. See `tools/spf/README.spf` in the source tree for a detailed manual.

1.8.7 syscall

[TDB.]

1.8.8 cfi_check

[TDB.]

1.8.9 rop_checkret

[TDB.]

1.9 Examples and Demos

Before you begin writing Stackdb programs yourself, you'll want to get something working first. Besides, some Stackdb tools may be generally useful to you. This section provides some quick examples of using some of Stackdb's tools so you quickly use its core functionality. When you start building your own tools, these may provide helpful insights.

1.9.1 Preparing Your System

If you want to run any of the Xen or QEMU/KVM demos, please make sure you've built Stackdb with the required support, and have installed Xen and/or QEMU. Furthermore, please download the (large) tarball at http://www.flux.utah.edu/software/stackdb/downloads/centos5.5-x86_64.tar.gz (about 275MB, currently). Create a directory on your machine, and extract the tarball into it:

```
$ tar -xzv --strip-components=1 \
      -f vm-2.6.18-files.tar.gz -C /path/to/your/directory
```

This directory contains a 2.6.18 kernel; two initramfs images; and debuginfo files for nearly all the binaries in the initramfs. You'll be able to use this directory in the Xen and/or QEMU/KVM examples. The only user with a passwd is `root`; the passwd is also `root`.

1.9.2 KVM demos

First, let's do something simple. Get your system prepared to run KVM VMs in the manner required by Stackdb, as described above. Then start up a KVM VM:

```
$ sudo QEMU_MEMPATH_PREFIX=/hugetlbfs/qemu \
       LD_PRELOAD=/home/johnsond/g/a3/vmi.obj.tap.current/target/.libs/libqemuhacks.so.0.0.0
    qemu-system-x86_64 -cpu host -m 512 -enable-kvm \
      -kernel /tftpboot/roots/centos5.5-x86_64/boot/vmlinuz-2.6.18-308.el5 \
      -initrd /tftpboot/roots/centos5.5-x86_64/boot/initrd-2.6.18-308-full.el5.img \
      -append console=ttyS0 -nographic \
      -device e1000,netdev=net0 -netdev tap,id=net0 \
      -gdb tcp:127.0.0.1:1234,nowait,nodelay,server \
      -qmp tcp:127.0.0.1:1235,server,nowait -mem-path /hugetlbfs
```

This will give you a VM with the console attached to your terminal's stdio, and an e1000 network adapter (if you want your device connected to anything, make sure to read http://www.linux-kvm.org/page/Networking).

(Note that you'll need to set the LD_PRELOAD path to libqemuhacks.so.0.0.0 to point to your build tree, not mine! Also change the -kernel and -initrd arguments to point to wherever you unpacked the tarball, not `/tftpboot/roots/centos5.5-x86_64`.)

(Note also that if you are attaching to a QEMU VM that is managed by `libvirt`, or a cloud platform such as OpenStack or Eucalyptus, you'll want to read the [GDB/QEMU/KVM driver notes].)

Once it's booted, login with user `root`, passwd `root`.

Second, make sure you've installed Stackdb. Then, run the `dumptarget` program in another terminal like this:

```
$ sudo dumptarget -t gdb \
      --gdb-port 1234 --qemu --qemu-qmp-port 1235 --kvm \
      -M /tftpboot/roots/centos5.5-x86_64/boot/vmlinux-syms-2.6.18-308.el5 \
      --qemu-mem-path /hugetlbfs/qemu_back_mem.pc.ram.XXXXXX \
  --personality linux -s sys_open
```

(and replace XXXXXX with the path to your QEMU process's RAM file on the hugetlbfs mountpoint; this will change each time you run your QEMU VM!).

Finally, go back to the VM's console, and type `ls`. In the terminal you started `dumptarget` in, you should see output like

```
bsymbol (region(main:/tftpboot/roots/centos5.5-x86_64/boot/vmlinux-syms-2.6.18-308.el5))
  lsymbol:
    func: long int sys_open (const char* filename (line=1179,) @@ LIST([0xffffffff80031312,(
Starting main debugging loop!
sys_open (0xffffffff80031312) (thread 616) mode = 1 (0x01000000), filename = "/etc/ld.so.cac
sys_open (0xffffffff80031312) (thread 616) mode = -976703488 (0x00b0c8c5), filename = "/lib6
sys_open (0xffffffff80031312) (thread 616) mode = -976703488 (0x00b0c8c5), filename = "/lib6
sys_open (0xffffffff80031312) (thread 616) mode = -976703488 (0x00b0c8c5), filename = "/lib6
sys_open (0xffffffff80031312) (thread 616) mode = -976703488 (0x00b0c8c5), filename = "/lib6
sys_open (0xffffffff80031312) (thread 616) mode = -976699152 (0xf0c0c8c5), filename = "/lib6
sys_open (0xffffffff80031312) (thread 616) mode = -976697928 (0xb8c5c8c5), filename = "/lib6
sys_open (0xffffffff80031312) (thread 616) mode = -976696704 (0x80cac8c5), filename = "/lib6
sys_open (0xffffffff80031312) (thread 616) mode = -976696704 (0x80cac8c5), filename = "/lib6
sys_open (0xffffffff80031312) (thread 616) mode = 438 (0xb6010000), filename = "/proc/mounts
sys_open (0xffffffff80031312) (thread 616) mode = 0 (0x00000000), filename = "/selinux/mls"
sys_open (0xffffffff80031312) (thread 616) mode = 1 (0x01000000), filename = "." (0x2e00), f
```

You can see dumptarget successfully found the `sys_open` symbol, and placed a probe on it, and resumed the VM.

Then, once I typed `ls` on the VM's serial console, many `sys_open` system calls occurred. You can see which ones yourself by looking at the output above.

Try changing sys_open to another function you're interested in.

1.9.3 Ptrace demos

An easy to way to see the Ptrace backend in action is to simply start up a couple of the test programs that come with Stackdb. These programs don't get built unless you specified `--enable-tests` to `configure`. However, if you return to the Stackdb build directory, and type `make -C tests`, several test programs will build (don't worry if the build ends in a failure—the programs you want, `dummy` and `dummy.threads` should still be there). These programs simply loop while updating some counter variables via a tail-call stack of functions (`f1` to `f10`) and sleep for a few seconds. `dummy` does this with a single thread; `dummy.threads` does this with two threads.

First, in one terminal, fire of `dummy`. Then, in another terminal, run some of the Stackdb tools against it:

```
$ sudo ./backtrace -p `pgrep -n dummy`
```

(Recall, `pgrep` looks for a process ID whose name matches the argument supplied.) You should see output like this:

```
Initial threads in target 'ptrace(21424)':
tid(21424): tid=21424,name=(null),curctxt=0,ptid=-1,uid=-1,gid=-1,
  ip=7f19264a3920,bp=ffffffff,sp=7fff37896ba8,flags=246,ax=ffffffffffffffdfc,
  bx=7fff37896bc0,cx=ffffffffffffffff,dx=0,di=7fff37896bb0,si=7fff37896bb0,
  cs=33,ss=2b,ds=0,es=0,fs=0,gs=0
tid(21424): tid=21424,name=(null),curctxt=0,ptid=-1,uid=-1,gid=-1,
  ip=7f19264a3920,bp=ffffffff,sp=7fff37896ba8,flags=246,ax=ffffffffffffffdfc,
  bx=7fff37896bc0,cx=ffffffffffffffff,dx=0,di=7fff37896bb0,si=7fff37896bb0,
  cs=33,ss=2b,ds=0,es=0,fs=0,gs=0
thread 21424:
#0 0x00007f19264a3920 in ../sysdeps/unix/syscall-template.S ()
    at ../sysdeps/unix/syscall-template.S:81
#1 0x00007f19264a37e1 in __sleep (seconds=0)
    at ../nptl/sysdeps/unix/sysv/linux/sleep.c:-1
#2 0x0000000000400d47 in looper (istart=?)
    at ../../vmi.tap.current/tests/dummy.c:207
#3 0x00000000004008f0 in main (argc=40,argv=?)
    at ../../vmi.tap.current/tests/dummy.c:255
#4 0x00007f192640adb5 in __libc_start_main (main=0x400830,argc=4,
         argv="./dummy",init=?,fini=?,rtld_fini=?,stack_end=0x7fff37896ea8)
    at libc-start.c:319
#5 0x0000000000400925 in _start () at /proc/21424/exe:-1
ptrace(21424) finished.
```

If you run `dummy.threads` and run `backtrace` against that (i.e., `sudo ./backtrace -p`pgrep -n dummy.threads“), you'll see two thread backtraces.

1.10 Writing Stackdb Programs in C

The way to think about developing a program using Stackdb is that you're scripting a debugger-like (or memory forensics) interaction with a target program running on the same host, or on another host. You want to investigate it, either by passively reading its memory, or by actively installing breakpoints and watchpoints, and reacting to its control flows. Sometimes you might even alter memory content or change control flow by modifying CPU registers or the stack. Usually, after inserting some probes, you let the target program continue running, and “handle” the events when your target program hits the probes (perhaps you do some kind of analysis, or wait for a specific kind of state to be reached).

1.10.1 Library Overview

In a Stackdb program, the target program being debugged is accessed through a target "object," as illustrated in the Stackdb architecture diagram. A target object corresponds to a particular level of abstraction or a portion of the whole system being debugged; and it provides access to the target program and creates and maintains a model of it (i.e., tracks its threads, its address spaces, etc). By invoking target API functions, which are common to all targets, you can install breakpoints ("probes" in Stackdb terminology), examine software state and symbols, single-step, and potentially modify execution at the level of a particular target.

The Stackdb architecture diagram also shows that each target is paired with a driver, whose purpose is to implement debugger-like inspection and control features for a particular software abstraction: e.g., kernel, process, or language runtime. Although all drivers implement a common driver API, we distinguish two primary classes of implementation. A base driver interacts directly with the system being debugged, e.g., via a hypervisor-provided interface or `ptrace(2)`. An overlay driver interacts with the system through another target, i.e., by "stacking on top of an appropriate underlying target. The overlay driver communicates with the underlying target through the target API.

Because the target API is "implemented" by every target, a user can easily instantiate multi-level stacks of targets. In addition, the ability to implement drivers in terms of underlying targets greatly eases the process of developing new drivers, e.g., for new language runtimes. Finally, the target API makes it possible to implement generic analyses and utilities that can be applied to multiple levels of a software stack.

1.10.2 Integrating Stackdb Into Your Program: "Running" Stackdb

When you write a Stackdb program, you might want to insert some probes, wait until they are hit, and then analyze the target program's state, and/or modify its execution. In this mode, your program will be a simple event loop — you call a Stackdb function that "runs" your target program (`target_monitor()`) and calls your probe handlers when their corresponding probes are hit — Stackdb does the "dirty work" behind the scenes to implement breakpoints, safely read and write target memory and CPU registers, etc. Your handlers do all the work while the target program is paused. This is a simple, synchronous, event-driven model for writing Stackdb programs. Currently, only the base drivers need to be "run" — but the API supports runnable overlay drivers as well, if you need to develop some strange hybrid overlay driver that both receives exception notifications from its underlying base target, and from some external source (very unlikely).

If you are integrating Stackdb with another program that must do other tasks in

addition to handling debug exceptions, you'll need to choose between a couple different styles. (You'll also need to handle signals in your Stackdb program; if your program is signaled with a fatal signal, and you haven't specified a handler for the signal, your Stackdb program will exit before the Stackdb library can remove probes on the target! This likely will crash the target program next time it hits a breakpoint. See the section on signal handling below; Stackdb can help.)

First, you can use one of the Stackdb monitoring/polling mechanisms below to create your own event loop that can monitor one or more targets. If you have an existing event loop and don't want to poll, you can minimize polling overhead by adding Stackdb targets to your own event loop construct (which probably involves a `select()` loop — you might already have one, or might have to create one — Stackdb provides its own event loop abstraction around `select()` that you can borrow if you like). Most Stackdb drivers are able to proxy debug exceptions over a file descriptor — in other words, they write a single byte to a file descriptor when the target has a debug exception that needs to be handled — and that file descriptor can then be added to a `select()` loop. When the file descriptor is ready to read, then you should call the `target_poll()` function to handle the pending debug exception. Third, you can either divide your program into multiple threads or multiple processes; and dedicate a single thread or process to call `target_monitor()`. In this case, you'll need to use appropriate synchronization mechanisms to coordinate your threads' interoperation. Stackdb is *not thread-safe* — it does not have built-in support for a threading API like `pthreads(2)` — and it is never safe to apply the Target API to a single target from different threads at the same time. If you need to do this, you'll need to protect such accesses with a mutex of some kind!

Finally, you might want to analyze the target's memory asynchronously, without pausing the target (except when you attach or detach from it). In Stackdb, you can access memory asynchronously (without pausing the target program), but you must pause it to read or write its CPU state. Thus, in this mode, you might never pause or unpause the target — and never call `target_monitor()` nor `target_poll()`.

1.10.2.1 Monitoring One Target: `target_monitor()`

If you only need to monitor a single target object, and don't already have an existing program or event loop, you can use `target_monitor()`.

1.10.2.2 Monitoring Multiple Targets: `target_monitor_evloop()`

If you need to monitor multiple base target objects, you'll want to use `target_monitor_evloop()` (unless you have written a new base driver that doesn't support evloops!). This function uses Stackdb's evloops, and if you have existing file descriptor-based sources of events in a program you're adding

Stackdb to, you may be able to replace your existing event loop with Stackdb's evloop.

(*If you use Stackdb's evloops, be aware that some drivers may use the* `waitpipe` *data structure to turn signals into file descriptor I/O that may be detected via a select() loop; so don't install your own* `SIGCHLD` *handler!*)

1.10.2.3 Monitoring Multiple Targets (polling): `target_poll()`

If your application is best-suited to a polling style of handling Stackdb targets, you can use Stackdb's `target_poll()` function to check for (and handle) debug exceptions, and regularly poll in a coordinated manner with your program's other tasks. With polling, you can avoid blocking your program's main thread of control, or block for finite amounts of time.

1.10.2.4 Manually Handling One or More Targets: `evloop_run()`

(*This section is not for the faint-of-heart; it will cause your code to become more complex; make sure your use case really demands this power.*)

The `struct evloop` object is essentially a powerful wrapper around a select() loop. It provides a loop monitor that monitors a set of file descriptors with select(), and triggers callback functions associated with those descriptors when they are available for I/O. Thus, any target type whose debug exceptions can be represented by file I/O can be monitored by an evloop. All current built-in Stackdb drivers support evloops.

To use this run mechanism, call `evloop_create()` to obtain an evloop. Then, pass that evloop to each target_instantiate*() call you make; your new target will be attached to the evloop. Finally, call `evloop_run()` to run the loop. Read the evloop API documentation for more details, or take a look at `examples/multi-target-evloop.c`.

(*If you use evloops, be aware that some drivers may use the* `waitpipe` *data structure to turn signals into file descriptor I/O that may be detected via a select() loop; so don't install your own* `SIGCHLD` *handler!*)

1.10.2.5 Signals

Unfortunately, since Stackdb handles significant I/O on your behalf, and responds to asynchronous events like signals — you must carefully deal with signals.

Never send a fatal signal to a Stackdb program. For instance, when using GDB, if you have installed a software breakpoint in a program; then kill GDB with signal 9 or 15; then trigger the breakpoint to be hit in the target program — the program will die with an unhandled debug trap. The same is true is of Stackdb. If you signal a debugger program with a signal number it doesn't (or

can't) catch, any modifications that it made to its target will persist and very likely cause trouble!

Stackdb provides much built-in help to appropriately handle signals on your behalf. Its default signal handler (see target_install_default_sighandler(); target_default_sighandler() is the actual handler function) catches `SIGHUP`, `SIGINT`, `SIGQUIT`, `SIGABRT`, `SIGFPE`, `SIGSEGV`, `SIGPIPE`, `SIGALRM`, `SIGTERM`. Only `SIGINT` is nonfatal (it will cause whatever target_monitor*() function that is running the loop to return with an interrupt condition), so you can do some asynchronous work, like handling user input. The other signals cause an immediate target_pause(), target_close(), and target_finalize() to be applied to each existing target.

However, Stackdb does not provide a general signal-handling and -dispatch infrastructure. Odds are that if you need such support, you'll want to design your own signal handling mechanisms, and manually clean up all Stackdb targets (i.e., close and finalize) as necessary.

There's one other important note about Stackdb's signal usage. Some Stackdb drivers make use of the internal `waitpipe` object, which turns a `SIGCHLD` into a write on a pipe file descriptor (so that a select loop can be notified — this is how we turn all debug exception notifications into file descriptor events). If you need to handle `SIGCHLD` signals yourself, make sure to provide your own handler to `waitpipe_init_ext()`. If this mechanism isn't sufficient (i.e., you need fine-grained control over the `sigaction()` mask or flags), let us know.

1.10.2.6 Forking

Stackdb does not support use of fork() at the moment. If you must embed Stackdb in a multi-process architecture, do not call fork() once you've instantiated Stackdb targets within a process.

1.10.3 Targets

[TBD.]

1.10.3.1 CPU State: Threads and Registers

[TBD.]

1.10.3.2 Memory: Address Spaces, Regions, Ranges

[TBD.]

1.11 Overlay Targets

[TBD.]

1.11.1 Examples

Let's walk through some examples. The full source code for each of these programs can be found in the source tree in the `examples/` subdirectory.

1.11.1.1 Example 1: Placing a Breakpoint in a Userspace Process

Let's start simply

1.11.1.2 Example 3: Finding Unique Control Flows in the Linux Kernel

Suppose we want to write a program that allows us to find unique control flows at known, interesting functions within another program. For instance, maybe we want to figure out who calls the `ip_rcv` function in the Linux kernel. Even though `ip_rcv` is only called through a function pointer table in the kernel's IP subsystem, we could probably figure this out through some static analysis—but this is more fun!

Our strategy will be to set a breakpoint on the `ip_rcv` function; generate a backtrace as a string; and hash it into a hashtable that maintains a count. However, the Stackdb backtrace string-ifier is fairly powerful, and the backtraces we get will probably be fairly unique, due to function call arguments. So we'll actually generate the backtrace once, and stringify it twice — once without any function arguments; and once with them.

First, we'll start by including the primary header files we need:

```
#include "common.h"
#include "target_api.h"
#include <glib.h>
```

`common.h` contains some basic types, such as the arch-independent `ADDR` numeric type. `target_api.h` contains the user-available functions that allow you to debug and analyze targets. We'll use `glib` for hashtable functionality.

Next, we'll write a simple main that looks up a symbol and places a probe (a breakpoint or watchpoint) on it. We'll also write a simple callback function to handle the probe event; the Stackdb library notifies your program on probe exceptions via callbacks. Stackdb handles the work of pausing and unpausing the target program for you, and manages its execution for you—so all you have to know is that the target is stopped whenever your callbacks are called.

```
GHashTable *ht;

int main(int arg,char **argv) {
    struct target *t;
    struct bsymbol *myfunc;

    /* Initialize the core Stackdb globals */
    target_init();
    atexit(target_fini));

    /* Create a string hashtable */
    ht = g_hash_table_new_full(g_str_hash,g_str_equal,NULL,free);

    /* Parse the command-line arguments */
    tspec = target_argp_driver_parse(NULL,NULL,argc,argv,
                                     TARGET_TYPE_ALL_BASE,1);
    if (!tspec) {
        fprintf(STDERR,"ERROR: could not parse target arguments!\n");
        exit(-1);
    }

/* Instantiate a target */
```

Now, let's make it possible to pass a list of functions on the command line, all of which we probe for unique execution paths:

Now let's add an option to disable certain threads based on name:

[TBD...]

1.11.2 C API

Refer to the online Stackdb C API.

1.12 Writing Stackdb Programs in Python

We haven't yet built this binding, but may do so.