

Systemtap tutorial

Frank Ch. Eigler fchere@redhat.com
February 28, 2025

Contents

1	Introduction	2
2	Tracing	2
2.1	Where to probe	3
2.2	What to print	4
2.3	Exercises	4
3	Analysis	5
3.1	Basic constructs	6
3.2	Target variables	6
3.3	Functions	7
3.4	Arrays	7
3.5	Aggregates	
3.6	Safety	10
3.7	Exercises	10
4	Tapsets	11
4.1	Automatic selection	11
4.2	Probe point aliases	12
4.3	Embedded C	13
4.4	Naming conventions	15
4.5	Exercises	15
5	Further information	15
A	Errors	16
A.1	Parse errors	16
A.2	Type errors	16
A.3	Symbol errors	16
A.4	Probing errors	17
A.5	Runtime errors	17

B Acknowledgments

Introduction

Systemtap is a tool that allows developers and administrators to write and reuse simple scripts to deeply examine the activities of a live Linux system. It may be effectively filtered and summarized quickly and safely to enable diagnoses of complex performance or functional problems.

NTE This tutorial does not describe every feature available in systemtap. Please see the individual stap manual pages for the most up-to-date information. These may be available installed on your system or at <https://sourceware.org/systemtap/man>.

The essential idea behind a systemtap script is to name events and to give them handlers. Whenever a specified event occurs the Linux kernel runs the handler as if it were a quick subroutine then resumes. There are several kinds of events such as entering or exiting a function, a timer expiring, or the entire systemtap session starting or stopping. A handler is a series of script language statements that specify the work to be done whenever the event occurs. This work normally includes extracting data from the event context, storing them into internal variables, or printing results.

Systemtap works by translating the script to C, running the system C compiler to create a kernel module from that. When the module is loaded, it activates all the provided events by hooking into the kernel. Then, as events occur on any processor, the compiled handlers run. Eventually, the session stops, the hooks are disconnected, and the module removed. This entire process is driven from a single command-line program, stap.

```
cat helloworld.stp
probe begin

    print hello world\n
    exit

stap helloworld.stp
hello world
```

Figure A systemtap smoke test.

This paper assumes that you have installed systemtap and its prerequisite kernel development tools and debugging data so that you can run the scripts such as the simple one in Figure A. Log on as root or even better, log in as a user that is a member of the stapdev group or as a user authorized to sudo before running systemtap.

Tracing

The simplest kind of probe is simply to trace an event. This is the effect of inserting strategically located print statements into a program. This is often the first step of problem solving, explore by seeing a history of what has happened.

This style of instrumentation is the simplest. It just asks systemtap to print something at each event. To express this in the script language, you need to say where to probe and what to print there.

```

cat strace-open.stp
probe sscall.open

printf      s      open      s n      eecnae      pi      argstr

probe tier.s      after      secons

eit

stap strace-open.stp
ware-guest      open      /etc/rehat-release
hal      open      /e/hc
hal      open      /e/hc
hal      open      /e/hc
f      open      /etc/l.so.cache
f      open      /lib/tls/libc.so.
f      open      /etc/tab
hal      open      /e/hc

```

Figure 1: A taste of sstetap: a sste-wie strace ust for theopen sste call.

1. here to probe

stetap supports a nuber of built-in eents. he librar of scripts that coes with sstetap each calle a tapset a e ne aitional ones e ne inters of the b uilt-in faillee the stapprobes an page for etails on these an an other probe point failies. ll these eents are nae using a uni e snta with ot-separate paraeterize ienti ers:

	begin	he startup of the sstetap session.
	en	he en of the sstetap session.
ernel.function	ssopen	he entr to the function nae ssopen in the ernel.
	sscall.close.return	he return fro the close sste call.
oule et .stateent	eabeef	he aresse instruction in the et lesste rier.
	tier.s	tier that res eer illisecons.
	tier.profile	tier that res perioicall on eer .
	perf.hw.cacheisses	particular nuber of cache isses hae occurre.
	procfs status .rea	process tring to rea a snthetic le.
process a.out .stateent	ain.c:	ine of the a.out progra.

et s sa that ou woul lie to trace all function entries in a source le sa net/socet.c in the ernel. he ernel.function probe point lets ou epress that easil since sstetap aines the ernel s ebugging inforation to relate obect coe to source coe.t wors lie a ebugger: if ou can nae or place it ou can probe it. se ernel.function net/socet.c .call for the function entries an ernel.function net/socet.c .return for atching eits. ote the use of wilcars in the function nae part an the subseuent part. ou can also put wilcars into the le nae an een a a colon : an a line nuber if ou want to restrict the search that pisel. ince sstetap will put a separate probe in eer place that atches a probe point a few wilcars can epan to hunres or thousands of probes so be careful what ou as for.

ithout the .call uali er inline function instances are also probe but hae no corresponing .return.

Once you identify the probe points, the skeleton of the `systemtap` script appears. The `probe` keyword introduces a probe point, or a comma-separated list of them. The `{}` and `do { }` braces enclose the handler for all listed probe points.

```
probe kernel.functionnet/socket.c
probe kernel.functionnet/socket.c{return
```

You can run this script as is, though with empty handlers there will be no output. Put the two lines into a new file. Run `systemtap -v FILE`. Terminate it any time with `C`. The `-v` option tells `systemtap` to print more verbose messages during its processing. Try the option to see more options.

2.2 What to print

Since you are interested in each function that was entered and exited, a line should be printed for each, containing the function name. In order to make that list easy to read, `systemtap` should indent the lines so that functions called by other traced functions are nested properly. To tell each single process apart from any others that may be running concurrently, `systemtap` should also print the process ID in the line.

`Systemtap` provides a variety of such contextual data, ready for formatting. They usually appear as function calls within the handler, like you already saw in Figure 5. See the function:: man pages for those functions and more defined in the `tapset` library, but here's a sampling:

<code>tid</code>	The id of the current thread.
<code>pid</code>	The process task group id of the current thread.
<code>uid</code>	The id of the current user.
<code>execname</code>	The name of the current process.
<code>cpu</code>	The current cpu number.
<code>gettimeofday</code>	Number of seconds since epoch.
<code>getcycles</code>	Snapshot of hardware cycle counter.
<code>pp</code>	A string describing the probe point being currently handled
<code>ppfunc</code>	If known, the the function name in which this probe was placed
<code>vars</code>	If available, a pretty-printed listing of all local variables in scope.
<code>printbacktrace</code>	If possible, print a kernel backtrace.
<code>printubacktrace</code>	If possible, print a user-space backtrace.

The values returned may be strings or numbers. The `print` built-in function accepts either as its sole argument. Or, you can use the C-style `printf` built-in, whose formatting argument may include `%s` for a string, `%d` for a number. `printf` and other functions take comma-separated arguments. Don't forget an `nl` at the end. There exist more printing / formatting functions.

A particularly handy function in the `tapset` library is `threadindent`. Given an indentation delta parameter, it stores internally an indentation counter for each thread, and returns a string with some generic trace data plus an appropriate number of indentation spaces. The generic data includes a timestamp number of microseconds since the initial indentation for the thread, a process name and the thread id itself. It therefore gives an idea not only about what functions were called, but who called them, and how long they took. Figure 3 shows the finished script. It lacks a call to the `exit` function, so you need to interrupt it with `C` when you want the tracing to stop.

2.3 Exercises

1. Use the `-l` option to `systemtap` to list all the kernel functions named with the word `init` in them.

```

cat socket-trace.stp
probe kernel.functionnet/socket.c.call
    printf s - sn, threadindent1, ppfunc

probe kernel.functionnet/socket.c.return
    printf s - sn, threadindent-1, ppfunc

stap socket-trace.stp
    0 hald2632: - sockpoll
    28 hald2632: - sockpoll
...
    0 ftp7223: - syssocketcall
    115 ftp7223: - syssocket
    2173 ftp7223: - sockcreate
    2286 ftp7223: - sockallocinode
    2737 ftp7223: - sockallocinode
    334 ftp7223: - sockalloc
    338 ftp7223: - sockalloc
    3417 ftp7223: - sockcreate
    4117 ftp7223: - sockcreate
    4160 ftp7223: - sockcreate
    4301 ftp7223: - sockmapfd
    4644 ftp7223: - sockmapfile
    46 ftp7223: - sockmapfile
    4715 ftp7223: - sockmapfd
    4732 ftp7223: - syssocket
    4775 ftp7223: - syssocketcall
...

```

Figure 3: Tracing and timing functions in net/sockets.c .

2. Trace some system calls using `syscall.NAME` and `.return` probe points, with the same `threadindent` probe handler as in Figure 3. Print parameters using `ppargs` and `return` . Interpret the results.
3. Change figure 3 by removing the `call` modifier from the first probe. Note how function entry and function return now don't match anymore. This is because the first probe will match both normal function entry and inlined functions. Try putting the `call` modifier back and add another probe using `probe kernel.functionnet/socket.c.inline` . What `printf` statement can you come up with in the probe handler to show the inlined function entry precisely in between the `call` and `.return` thread indented output?

3 Analysis

Pages of generic tracing text may give you enough information for exploring a system. With `systemtap`, it is possible to analyze that data, to filter, aggregate, transform and summarize it. Different probes can work together to share data. Probe handlers can use a rich set of tool constructs to describe algorithms, with a syntax taken roughly from `awk`. With these tools, `systemtap` scripts can focus on a specific question and provide a compact response: `grep` needed.

. Basic constructs

Most systemtap scripts include conditionals to limit tracing or other logic to those processes or users or whatever of interest. The syntax is simple

```
if  EXPR STATEMENT else STATEMENT  ifelse statement
while  EXPR STATEMENT  while loop
for  A B C STATEMENT  for loop
```

Scripts may use `break` `continue` as in C. Probe handlers can return early using `net` as in awk. Blocks of statements are enclosed in `{}` and `}`. In systemtap the semicolon `;` is accepted as a null statement rather than as a statement terminator so `;` is only rarely necessary. Shellstyle `/* */` `Cstyle` `/* */` and `Cstyle` comments are all accepted.

Expressions look like C or awk and support the usual operators precedences and numeric literals. Strings are treated as atomic values rather than arrays of characters. String concatenation is done with the dot `a . b`. Some examples

```
uid          probably an ordinary user
eecname sed   current process is sed
cpu  gettimeofday  after eb. on CP
hello . . world  a string in three easy pieces
```

Variables may be used as well. Just pick a name assign to it and use it in expressions. They are automatically initialized and declared. The type of each identifier string vs. number is automatically inferred by systemtap from the kinds of operators and literals used on it. Any inconsistencies will be reported as errors. Conversion between string and number types is done through explicit function calls.

```
foo  gettimeofday  foo is a number
bar  usrbins . eecname  bar is a string
c    c is a number
s    sprintf        s becomes the string
```

By default variables are local to the probe they are used in. That is they are initialized used and disposed of at each probe handler invocation. To share variables between probes declare them global anywhere in the script. Because of possible concurrency multiple probe handlers running on different CPUs each global variable used by a probe is automatically read or written while the handler is running.

. Target variables

A class of special target variables allow access to the probe point context. In a symbolic debugger when you're stopped at a breakpoint you can print values from the programs context. In systemtap scripts for those probe points that match with specific executable point rather than an asynchronous event like a timer you can do the same.

In addition you can take their address the `&` operator prettyprint structures the `pprint` and `pprint` pretty print multiple variables in scope the `vars` and related variables or cast pointers to their types the `cast` operator or test their existence resolvability the `defined` operator. Read about these in the manual pages.

To know which variables are likely to be available you will need to be familiar with the kernel source you are probing. In addition you will need to check that the compiler has not optimized those values into unreachable nonexistence. You can use `stack PRBEPNT` to enumerate the variables available there.

se them between consecutive expressions that place unary or mixed prepost in an ambiguous manner.

```

cat timeriffies.stp
global countiffies countms
probe timer.iffies countiffies
probe timer.ms countms
probe timer.ms

h countiffies countms
printf iffiesms ratio dd CNdn
countiffies countms h
eit

stap timeriffies.stp
iffiesms ratio CN

```

Figure Experimentally measuring CN - .

ets say that you are trying to trace filesystem readwrite s to a particular deviceinode. rom your knowl edge of the kernel you know that two functions of interest co uld be vfsread and vfwrite . Each takes a struct file argument inside there is either a struct dentry or struct path which has a struct dentry . The struct dentry contains a struct inode and so on. Systemtap allows limited dereferencing of such pointer chains. Two functions userstring and kernelstring can copy char target variables into systemtap strings. igure demonstr ates one way to monitor a particular file identified by device number and inode number. The script selects the ap propriate variants of devnr andinodnr based on the kernel version. This eample also demonstrates passing numeric commandline arguments etc. into scripts.

. unctions

unctions are conveniently packaged reusable software it would be a shame to have to duplicate a comple condition epression or logging directive in every placed i ts used. So systemtap lets you define functions of your own. ike global variables systemtap functions may be defined anywhere in the script. They may take any number of string or numeric arguments by value an d may return a single string or number. The parameter types are inferred as for ordinary variables and must be consistent throughout the program. Local and global script variables are available but target variables are not. Thats because there is no specific debugginglevel contet associated with a function.

A function is defined with the keyword function followed by a name. Then comes a commaseparated formal argument list ust a list of variable names. The enclosed body consists of any list of statements including epressions that call functions. Recursion is po ssible up to a nesting depth limit. igure displays function synta.

. Arrays

ften probes will want to share data that cannot be represen ted as a simple scalar value. Much data is naturally tabular in nature indeed by some tuple of thread numbers processor ids names time and so on. Systemtap offers associative arrays for this purpose. T hese arrays are implemented as hash tables with a mainum si e that is fied at startup. Because they are too la rge to be created dynamically for individual probes handler runs they must be declared as global.

```

cat inode-watch.stp
probe kernel.function (vfswrite),
    kernel.function (vfsread)

if (defined(file-fpath-dentry))
    devnr  file-fpath-dentry-dinode-isb-sdev
    inodenr file-fpath-dentry-dinode-iino
else
    devnr  file-fdentry-dinode-isb-sdev
    inodenr file-fdentry-dinode-iino

if (devnr (1 20 2) major/minor device
    && inodenr 3)
    printf (s(d) s 0xx/un,
        execname(), pid(), ppfunc(), devnr, inodenr)

stat -c D i /etc/crontab
fd03 133099
stap inode-watch.stp 0xfd 3 133099
more(30789) vfsread 0xfd00003/133099
more(30789) vfsread 0xfd00003/133099

```

Figure 5: Watching for reads/writes to a particular file.

```

Red Hat convention; see /etc/login.defs UIDMIN
function systemuidp (u) return u 500

kernel device number assembly macro
function makedev (major,minor) return major 20 minor

function tracecommon ()

    printf(d s(d), gettimeofdays(), execname(), pid()
        no return value necessary

function fibonacci (i)

    if (i 1) return 0
    else if (i 2) return 1
    else return fibonacci(i-1) + fibonacci(i-2)

```

Figure 6: Some functions of dubious utility.

```

global a    declare global scalar or array variable
global b[400] declare array, reserving space for up to 400 tuples

```

The basic operations for arrays are setting and looking up elements. These are expressed in awksyntax: the

array name followed by an opening bracket a comma-separated list of index expressions and a closing bracket. Each index expression may be string or numeric as long as it is consistently typed throughout the script.

<code>foo hello</code>	increment the named array slot
<code>processusage uidename</code>	update a statistic
<code>times tid getcycles</code>	set a timestamp reference point
<code>delta getcycles times tid</code>	compute a timestamp delta

Array elements that have not been set may be fetched and return a dummy null value or an empty string as appropriate. However assigning a null value does not delete the element an explicit delete statement is required. Systemtap provides syntactic sugar for these operations in the form of explicit membership testing and deletion.

<code>if hello in foo</code>	membership test
<code>delete timetid</code>	deletion of a single element
<code>delete times</code>	deletion of all elements

The final and important operation is iteration over arrays. This uses the keyword `foreach`. Like `awk` this creates a loop that iterates over key tuples of an array not just values. In addition the iteration may be sorted by any single key or the value by adding an extra `or` code.

The `break` and `continue` statements work inside `foreach` loops too. Since arrays can be large but probe handlers must not run for long it is a good idea to exit iteration early if possible. The `limit` option in the `foreach` expression is one way. `simplicity` systemtap forbids any modification of an array while it is being iterated using a `foreach`.

<code>foreach ab in foo</code>	simple loop in arbitrary sequence
<code>foreach ab in foo limit</code>	loop in increasing sequence of value stop after
<code>foreach ab in foo</code>	loop in decreasing sequence of first key

2. Aggregates

When we said above that values can only be strings or numbers we lied a little. There is a third type statistics aggregates or aggregates for short. Instances of this type are used to collect statistics on numerical values where it is important to accumulate new data quickly without exclusive locks and in large volume storing only aggregated stream statistics. This type only makes sense for global variables and may be stored individually or as elements of an array.

To add a value to a statistics aggregate systemtap uses the special operator `+=`. Think of it like C's `output streamer` the left hand side object accumulates the data sample given on the right hand side. This operation is efficient taking a shared lock because the aggregate values are kept separately on each processor and are only aggregated across processors on request.

```
a += timestamp
writeuidname count
```

To read the aggregate value special functions are available to extract a selected statistical function. The aggregate value cannot be read by simply naming it as if it were an ordinary variable. These operations take an exclusive lock on the respective globals and should therefore be relatively rare. The simple ones are `min` `max` `count` `avg` and `sum` and evaluate to a single number. In addition histograms of the data stream may be extracted using the `histlog` and `histlinear`. These evaluate to a special sort of array that may

at present onl be printe.

```

                                ag a      the aerage of all the alues accuulate into a
print histlinear a              print an  ascii art  linear histogra of the sae ata
                                strea bouns  : : :  bucet with is
                                count writes zsh  the nuber of ties  zsh  ran the probe hanler
print histlog writes zsh        print an  ascii art  logarithic histogra of the sae
                                ata strea
```

. afet

he full epressiit of the scripting language raises ~~g~~uestions of safet. ere is a set of :

hat about in nite loops recursion probe hanler is boune in tie. he coe generate b
sstetap inclues eplicit checs that liit the total nuber of stateents eecute to a sall nuber.
siilar liit is ipose on the nesting epth of function calls. hen either liit is eceee that
probe hanler cleanl aborts an signals an error. he sstetap session is norall con gure to
abort as a whole at that tie.

hat about running out of eor o naic eor allocation whatsoeer taes place uring the
eecution of probe hanlers. rras function contets an buffers are allocate uring initialization.
hese resources a run out uring a session an generall r esult in errors.

hat about locing f ultiple probes see con icting locs on the sae global ariables one or ore
of the will tie out an be aborte. uch eents are tallie as sippe probes an a count is
isplae at session en. con gurable nuber of sippe pr obes can trigger an abort of the session.

hat about null pointers iision b zero he coe generate b sstetap translates potentiall
angerous operations to routines that chec their argu~~ent~~at run tie. hese signal errors if the are
inali. an arithetic an string operations silentl oer ow if the results ecee representation
liits.

hat about bugs in the translator copiler hile bugs in the translator or the runtie laer certainl
eist our test suite gies soe assurance. lus the entire ~~gene~~re coe a be inspecte tr the
-p option . opiler bugs are unliel to be of an greater concern for sstetap than for the ernel
as a whole. n other wors if it was reliable enough to buil~~le~~ternel it will buil the sstetap
oules properl too.

s that the whole truth n practice there are seeral wea points in sstetap an the unerling
probes sste at the tie of writing. utting probes inis~~cr~~iinatel into unusuall sensitie parts
of the ernel low leel contet switching interrupt is~~ph~~ing has reportel cause crashes in the
past. e are ing these bugs as the are foun an construct ing a probe point bloclist but it is
not coplete.

. ercises

. lter the last probe imier-iffies.stp to reset the counters an continue reporting instea of
eiting.

e anticipate support for ineing an looping using foreach shortl.
ee <http://sourceware.org/bugzilla>

- write a script that every ten seconds displays the top five most frequent users of open system call during that interval.
- write a script that experimentally measures the speed of the getcycles counter on each processor.
- use any suitable probe point to get an approximate profile of process CP usage which processes users use how much of each CP.

Tapsets

After writing enough analysis scripts for yourself you may become known as an expert to your colleagues who will want to use your scripts. Systemtap makes it possible to share in a controlled manner to build libraries of scripts that build on each other. In fact all of the functions `pid` etc. used in the scripts above come from tapset scripts like that. A tapset is just a script that designed for reuse by installation into a special directory.

Automatic selection

Systemtap attempts to resolve references to global symbols probes functions variables that are not defined within the script by a systematic search through the tapset library for scripts that define those symbols. Tapset scripts are installed under the default directory named `usr/share/systemtap/tapset`. A user may give additional directories with the `R` option. Systemtap searches these directories for script `.stp` files.

The search process includes subdirectories that are specially listed for a particular kernel version and/or architecture and ones that name only larger kernel families. Naturally the search is ordered from specific to general as shown in figure .

```

stap p v v e probe begin devnull
Created temporary directory tmpstaplNEBh
Searched usr/share/systemtap/tapset..i.          stp match count
Searched usr/share/systemtap/tapset...stp        match count
Searched usr/share/systemtap/tapset.i.stp        match count
Searched usr/share/systemtap/tapset..stp mat      ch count
Searched usr/share/systemtap/tapseti.stp ma      tch count
Searched usr/share/systemtap/tapset.stp match co  unt
Pass parsed user script and library scripts in usr      sysreal ms.
Running rm -rf tmpstaplNEBh

```

Figure illustrating the tapset search path.

When a script file is found that defines one of the undefined symbols that entire file is added to the probing session being analyzed. This search is repeated until no more references can become satisfied. Systemtap signals an error if any are still unresolved.

This mechanism enables several programming idioms. First it allows some global symbols to be defined only for applicable kernel version/architecture pairs and cause an error if their use is attempted on an inapplicable host. Similarly the same symbol can be defined differently depending on kernels in much the same way that different kernel includeasm/ARC files contain macros that provide a porting layer.

Another use is to separate the default parameters of a tapset consider a tapset that defines code for relating elapsed time data collection code can be generic with respect to which time unit it can use. It should have a default but should not require another. Figure shows a way.

routine from its implementation. For example intervals to process scheduling activities. The unit differs wallclock seconds cycle counts additional runtime checks to let a user choose

```
cat tapsettimecommon.stp
global timevars
function timerbegin name timevarsname time value
function timerend name return timevalue time varsname

cat tapsettimedefault.stp
function timevalue return gettimeofdayus

cat tapsettimeuser.stp
probe begin

timerbegin bench
for i i i
printf d cyclesn timerend bench
eit

function timevalue return getticks override f or greater precision
```

Figure Providing an overrideable default.

A tapset that reports only data may be as useful as ones that report functions or probe point aliases see below. Such global data can be computed and kept up to date using probes internal to the tapset. Any outside reference to the global variable would incidentally activate all the required probes.

Probe point aliases

Probe point aliases allow creation of new probe points from existing ones. This is useful if the new probe points are named to provide a higher level of abstraction. For example the systemcalls tapset defines probe point aliases of the form `syscall.open` etc. in terms of lower level ones like `kernel.functionsysopen`. Even if some future kernel renames `sysopen` the aliased name can remain valid.

A probe point alias definition looks like a normal probe. Both start with the keyword `probe` and have a probe handler statement block at the end. But where a normal probe just lists its probe points an alias creates a new name using the assignment operator. Another probe that names the new probe point will create an actual probe with the handler of the alias prepended.

This prepending behavior serves several purposes. It allows the alias definition to preprocess the content of the probe before passing control to the user-specified handler. This has several possible uses

- if flag flag net skip probe unless given condition is met
- name foo supply probedescribing values
- var var extract target variable to plain local variable

Figure demonstrates a probe point alias definition as well as its use. It demonstrates how a single probe point alias can expand to multiple probe points even to other aliases. It also includes probe point wildcarding. These functions are designed to compose sensibly.

```

cat probealias.stp
probe syscallgroup.io  syscall.open syscall.close
                        syscall.read syscall.write

groupname io

probe syscallgroup.process  syscall.fork syscall.eec          ve
groupname process

probe syscallgroup.
groups eecname . . groupname

probe end

foreach eg in groups
    printf s dn eg groupseg

global groups

stap probealias.stp
waitforsysio
udev.hotplugio
hal.hotplugio
Xio
apcsmartio
...
makeio
makeprocess
...
fcemcsmanageio
fdesktopio
...
mmsio
shio
shprocess

```

Figure 1. Classified system call activity.

1. Embedded C

Sometimes a tapset needs provide data values from the kernel that cannot be extracted using ordinary target variables `var`. This may be because the values are in complicated data structures may require lock awareness or are defined by layers of macros. Systemtap provides an escape hatch to go beyond what the language can safely offer. In certain contexts you may embed plain raw C in tapsets exchanging power for the safety guarantees listed in section 2. End-user scripts may not include embedded C code unless systemtap is run with the `getguru` mode option. Tapset scripts get `guru` mode privileges automatically.

Embedded C can be the body of a script function. Instead enclosing the function body statements in `use` and `end`. Any enclosed C code is literally transcribed into the kernel module it is up to you to make it safe and correct. In order to take parameters and return a value macros `STAPAR` and `STAPRETAE`

are made available. The familiar data-gathering functions `pid()` , `execname()`, and their neighbours are all embedded C functions. Figure 10 contains another example.

Since `systemtap` cannot examine the C code to infer these types, an optional⁵ annotation syntax is available to assist the type inference process. Simply suffix parameter names and/or the function name with `:string` or `:long` to designate the string or numeric type. In addition, the script may include a `block` at the outermost level of the script, in order to transcribe declarative code like `include linux/foo.h` . These enable the embedded C functions to refer to general kernel types.

There are a number of safety-related constraints that should be observed by developers of embedded C code.

1. Do not dereference pointers that are not known or testable valid.
2. Do not call any kernel routine that may cause a sleep or fault.
3. Consider possible undesirable recursion, where your embedded C function calls a routine that may be the subject of a probe. If that probe handler calls your embedded C function, you may suffer infinite regress. Similar problems may arise with respect to non-reentrant locks.
4. If locking of a data structure is necessary, use a `trylock` type call to attempt to take the lock. If that fails, give up, do not block.

```
cat embedded-C.stp

include linux/sched.h
include linux/list.h

function taskexecnamebypid:string (pid:long)
    struct taskstruct p;
    struct listhead p, n;
    listforeachsafe(p, n, &current-tasks)
        p listentry(p, struct taskstruct, tasks);
        if (p-pid (int)STAPARGpid)
            snprintf(STAPRETVALUE, MASTRINGLEN, s, p-comm);

probe begin

    printf(s(d)n, taskexecnamebypid(target()), ta          rget())
    exit()

pgrep emacs
16641
stap -g embedded-C.stp -x 16641
emacs(16641)
```

Figure 10: Embedded C function.

⁵This is only necessary if the types cannot be inferred from other sources, such as the call sites.

. Naming conventions

Since the tapset search mechanism just described potentially allows many script files to be selected for inclusion in a single session. This raises the problem of naming collisions where different tapsets accidentally use the same names for functions and globals. This can result in errors at translate or run time.

To control this problem systemtap tapset developers are advised to follow naming conventions. Here is some of the guidance.

- . Pick a unique name for your tapset and substitute it for `TAPSET` below.
- . Separate identifiers meant to be used by tapset users from those that are internal implementation artifacts.
- . Document the first set in the appropriate manpages.
- . Prefix the names of external identifiers with `TAPSET_` if there is any likelihood of collision with other tapsets or enduser scripts.
- . Prefix any probe point aliases with an appropriate prefix.
- . Prefix the names of internal identifiers with `__TAPSET__`.

. Exercises

- . Write a tapset that implements deferred and cancelable logging. Export a function that enqueues a tapset string into some private array returning an id token. Include a timer-based probe that periodically flushes the array to the standard log output. Export another function that if the entry was not already flushed allows a tapset string to be cancelled from the queue. You might speculate that similar functions and tapsets exist.
- . Create a relative timestamp tapset with functions that return all the same values as the ones in the timestamp tapset except that they are made relative to the start time of the script.
- . Create a tapset that exports a global array that contains a mapping of recently seen process numbers to process names. Intercept key system calls to update the list incrementally.
- . Send your tapset ideas to the mailing list

further information

For further information about systemtap several sources are available.

There are manpages

<code>stap</code>	systemtap program usage language summary
<code>stappaths</code>	your systemtap installation paths
<code>stapprobes</code>	probes probe aliases provided by builtin tapsets
<code>stape</code>	a few basic example scripts
<code>tapset</code>	summaries of the probes and functions in each tapset
<code>probe</code>	detailed descriptions of each probe
<code>function</code>	detailed descriptions of each function

There is much more documentation and sample scripts included. You may find them under `usr/share/doc/systemtap`.

Then there is the source code itself. Since systemtap is free software you should have available the entire source code. The source files in the `tapset` directory are also packaged along with the systemtap binary. Since systemtap reads these files rather than their documentation they are the most reliable way to see what's inside all the tapsets. Use the `-v` verbose command line option several times if you like to show inner workings.

Initially there is the project web site <https://sourceware.org/systemtap> with several articles and an archived public mailing list for users and developers [systemtap.sourceware.org](https://sourceware.org/systemtap/sourceware.org) IRC channels and a live T source repository. Come join us

A Errors

Explain some common systemtap error messages in this section. Most error messages include line character numbers with which one can locate the precise location of error in the script code. There is sometimes a subsequent or prior line that elaborates.

error at filename line column details

Many error messages contain a manpage key like this `man foo`. This indicates that more details are available as a manpage `foo` so use the `man foo` command to view it.

A. Parse errors

parse error expected foo saw bar

The script contained a grammar error. A different type of construct was expected in the given context.

parse error embedded code in unprivileged script

The script contained unsafe constructs such as embedded C sections but was run without the `-g` guru mode option. Confirm that the constructs are used safely then try again with `-g`.

A. Type errors

semantic error type mismatch for identifier foo string vs. long

In this case the identifier `foo` was previously inferred as a numeric type `long` but at the given point is being used as a string. Similar messages appear if an array index or function parameter slot is used with conflicting types.

semantic error unresolved type for identifier foo

The identifier `foo` was used for example in a `print` but without any operations that could assign it a type. Similar messages may appear if a symbol is misspelled by a typo.

semantic error Expecting symbol or array index expression

Something other than an assignable lvalue was on the left hand side of an assignment.

A. Symbol errors

while searching for arity N function semantic error unresolved function call

The script calls a function with `N` arguments that does not exist. The function may exist with different arity.

semantic error array locals not supported

An array operation is present for which no matching global declaration was found. Similar messages appear if an array is used with inconsistent arities.

semantic error variable foo modified during foreach

The array foo is being modified being assigned to or deleted from within a n active foreach loop. This invalid operation is also detected within a function called from within the loop.

A. Probing errors

semantic error probe point mismatch at position N while resolving probe point foo

A probe point was named that neither directly understood by systemtap nor defined as an alias by a tapset script. The divergence from the tree of probe point namespace is at position N starting with ero at left.

semantic error no match for probe point while resolving probe point foo

A probe point cannot be resolved for any of a variety of reasons. It may be a debuginfo-based probe point such as kernel.functionfoobar where no foobar function was found. This can occur if the script specifies a wildcard on function names or an invalid file name or source line number.

semantic error unresolved target symbol expression

A target variable was referred to in a probe handler that was not resolvable. A target variable is not valid at all in a context such as a script function. This variable may have been elided by an optimizing compiler or may not have a suitable type or there might just be an annoying bug somewhere. Try again with a slightly different probe point use statement instead of function to search for a more cooperative neighbour in the same area.

semantic error libdw failure

There was a problem processing the debugging information. It may simply be missing or may have some consistency/correctness problems. Later compilers tend to produce better debugging information so if you can upgrade and recompile your kernel/application it may help.

semantic error cannot find foo debuginfo

Similarly suitable debugging information was not found. Check that your kernel build/installation includes a matching version of debugging data.

A. Runtime errors

usually runtime errors cause a script to terminate. Some of these may be caught with the try ... catch ... construct.

ARNN Number of errors N skipped probes M

Errors and/or skipped probes occurred during this run.

division by

The script code performed an invalid division.

aggregate element not found

An statistics extractor function other than count was invoked on an aggregate that has not had any values accumulated yet. This is similar to a division by zero.

aggregation overflow

An array containing aggregate values contains too many distinct key tuples at this time.

MAXNESTING exceeded

Too many levels of function call nesting were attempted.

MAXACTION exceeded

The probe handler attempted to execute too many statements.

kernel/user string copy fault at 0xaddr

The probe handler attempted to copy a string from kernel or userspace at an invalid address.

pointer dereference fault

There was a fault encountered during a pointer dereference operation such as a target variable evaluation.

B Acknowledgments

The author thanks Martin Untch, Will Cohen, and Jim Eniston for improvement advice for this paper.