ExecVus — Execution Views

Alexandru Totolici
Department of Computer Science
University of British Columbia
201-2366 Main Mall
Vancouver BC Canada V6T 1Z4
alex.totolici@gmail.com

ABSTRACT

Operating systems are among some of the most complex forms of software developed for widespread use. This popularity causes most of the bugs to eventually bubble up and become action items for developers. The intricacies of any one operating system are rarely completely understood even by its developers, as code transcends programmer generations. Developers spend a long time before they can make the fix, first in reproducing a bug to identify where it occurs and second in understanding the code that executed the interesting path. ExecVus is a tool that helps developers visualize the characteristics of code that has been executed, in order to reduce the time cost of the latter task.

KEYWORDS: software visualization, text, source code.

1 INTRODUCTION

Complex software has complicated bugs and convoluted paths through which it reaches undesirable states. As practices go, simplifying software during development should be a goal for every programmer, but not much can be done about software that is already deployed and running. Some of the most important 'live' software is gigantic: operating systems have tens of millions of lines of source code (Mac OS X 10.4 had 86 million [1]), and they involve multiple generations of code, each written by a different developer. Whether it is Microsoft's proprietary Windows, with a team of senior software architects driving it, or open-source contender Linux, with a worldwide following of hackers tweaking it, no operating system has the luxury of developers knowledgeable in all the various subsystems and modules contained therein. Yet, when bugs are found and a fix is needed, it may not be the experts that those bugs go to, but developers moderately unfamiliar with the code.

Tralfamadore [2] reduces the time required to understand code by giving developers a way to look at what is essentially a recording (or a *trace*) of the execution of the entire system. The targeted operating system is run inside a modified version of the Qemu virtual machine emulator and Tralfamadore records all the processor operations, along with all the relevant information: where the call came from, where the next call happens, what data was used and by what parts of the code etc. Tralfamadore records what the CPU sees, but this view is not useful for developers looking to get a quick understanding of how a particular piece of code works: a quick peek at the trace data shows just how unfriendly it is to look at.

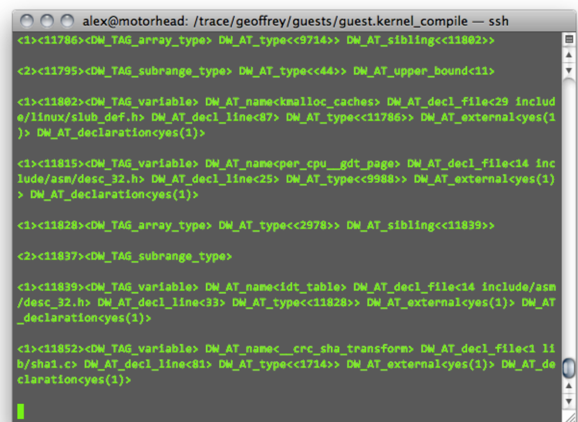

Figure 1. Contents of an ASCII-converted (raw) trace file, as seen in a shell session. Trace files are binary blobs parsed into ASCII text by Tralfamadore

The motivation behind Tralfamadore is not simply to collect this information and make it available to an analyst armed with text parsing command line tools, but to allow software developers to easily understand how code is executed. For that, we propose ExecVus, a visualization system that allows developers to get a quick understanding of code execution patterns.

The rest of this paper is as follows: In Section 2 we describe the key features of ExecVus. Following, in Section 3 we present two sample scenarios that illustrate how ExecVus could be used by developers. Section 4 looks at the related work in the area. Section 5 evaluates the strengths and weaknesses of ExecVus, while Section 6 presents avenues for future work and how ExecVus could be improved. The final section concludes.

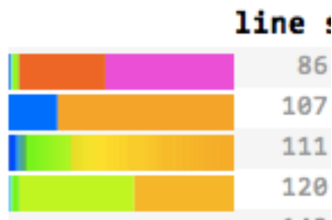

Figure 1. ExecVus in action in **fs/namei.c** Coloured bars are a way to visualize the available information.

ExecVus allows developers to see which code has been executed (for a given execution trace) and what calls have been dispatched by any given function. It uses the underlying data from Tralfamadore, though for this version we chose to use a pre-parsed trace. While this did not allow us to run any live queries, it made for faster application response as we did not have to wait on the processing tools to analyze the relevant portions of the trace. The annotations are shown overlaid on the Linux kernel source tree, as the trace data used had been recorded from a virtual machine running the Linux operating system.

There are two main components implemented in ExecVus currently. One is a collapsed view of the source code, showing only function headers and elements for which there is information in the trace. The second component adds scent to function names, giving developers a summary of the activity recorded for the function.

2.1 Code Collapse

A developer using ExecVus is not interested in the entirety of the source code, but only those sections that have been executed. As source files can be quite large and span multiple screens on a computer monitor, we decided to show, by default, only the lines of code that have interesting information associated. This includes symbol declarations, function headers to give developers an idea of which functions are used and which are not, and all lines for which trace data is available. Figure 2 shows the default view seen upon loading the
block/elevator.c file.

block/elevator.c

```
line source
  42 static DEFINE_SPINLOCK(elv_list_lock);
  43 static LIST_HEAD(elv_list);
  45 DEFINE_TRACE(block_rq_abort);
  50 static const int elv_hash_shift = 6;
  58 DEFINE_TRACE(block_rq_insert);
  59 DEFINE_TRACE(block_rq_issue);
  65 static int elv_iosched_allow_merge(struct request *rq, struct bio *bio)
  79 int elv_rq_merge_ok(struct request *rq, struct bio *bio)
 108   if (!elv_iosched_allow_merge(rq, bio))
 113 EXPORT_SYMBOL(elv_rq_merge_ok);
```

Figure 2. Default view of a source file, seen with ExecVus. Lines coloured in blue are considered interesting.

Function headers and symbol declarations are visible by default, in black. Lines of code coloured in blue represent elements for which the trace has additional information available. There are no restrictions imposed on what kind of elements these can be (i.e. function names), though of course comments could not have any execution information associated. Indeed, line 108 above is indented to show it belongs in the body of a function and we can tell it is an **if** statement.

By including the original source line numbers we are further emphasizing that this is a collapsed view and that gaps exist between the shown elements. Line numbering and code indentation are standard components of modern Integrated Development Environments (IDEs) that developers use routinely, and we felt that it is suitable to include them as a way to visually reinforce the purpose of ExecVus.

In case a developer requires it, the source code is still available and can be expanded quite easily by simply clicking on the name of a function. The source browser will slide the code into place, using animation for the transition. We felt that showing the code instantaneously would confuse the developer and create an unpleasant visual effect, and animating the transition was the natural solution to ensure continuity. The code slides down from the source of the click, with the direction of motion serving as a visual cue for the developer in case the action was triggered by an accidental click. Code without annotations, when expanded, is presented in a slightly faded shade of gray (Fig. 3) to distinguish it visually from the interesting sections.

```
115 static inline int elv_try_merge(struct request *__rq, struct bio *bio)
116 {
117   int ret = ELEVATOR_NO_MERGE;
118 
119   /*
120    * we can merge and sequence is ok, check if it's possible
121    */
122   if (elv_rq_merge_ok(__rq, bio)) {
123           if (__rq->sector + __rq->nr_sectors == bio->bi_sector)
124                   ret = ELEVATOR_BACK_MERGE;
125           else if (__rq->sector - bio_sectors(bio) == bio->bi_sector)
126                   ret = ELEVATOR_FRONT_MERGE;
127   }
128 
129   return ret;
130 }
```

Figure 3. Expanded function body

2.2 Scented Functions

The addition of scents to function names (in the form of horizontal, stacked bar graphs) gives users of ExecVus an immediate idea of the number of functions called by the scented function, and the relative distribution of these calls. Each colour band represents a function called by the scented function, while the area of the total bar that the band occupies indicates the percentage of calls, out of the total made by the calling function, that were directed to that particular callee.

The colours are computed in a way that gives the largest visual distribution, so that the differences between the bars are as obvious as possible, given a variable number of functions called.

```
644 void mntput_no_expire(struct vfsmo
646 repeat:
662 EXPORT_SYMBOL(mntput_no_expire);
664 void mnt_pin(struct vfsmount *mnt)
671 EXPORT_SYMBOL(mnt_pin);
673 void mnt_unpin(struct vfsmount *mr
683 EXPORT_SYMBOL(mnt_unpin);
685 static inline void mangle(struct s
696 int generic_show_options(struct se
711 EXPORT_SYMBOL(generic_show_options
726 void save_mount_options(struct sup
731 EXPORT_SYMBOL(save_mount_options);
733 void replace_mount_options(struct
742 EXPORT_SYMBOL(replace_mount_optior
746 static void *m_start(struct seq_fi
754 static void *m_next(struct seq_fil
761 static void m_stop(struct seq_file
766 struct proc_fs_info {
771 static int show_sb_opts(struct sec
789 static void show_mnt_opts(struct s
809 static void show_type(struct seq_f
818 static int show_vfsmnt(struct seq_
```

Figure 4. Scented function names. Colours represent various functions called within the body. Sizes are relative to the entire call count.

2.3 Technical Considerations

ExecVus is implemented on top of Mercurial[1]'s web view. As part of the original effort for Tralfamadore, modifications were made to the way Mercurial displays its information, namely adding code colouring and graphing control flow. Inline statistics were also provided for function calls, but they were only done by showing the number of times a path was taken, without any other visual cues.

Animations are implemented in JavaScript, with the aid of the jQuery[2] framework. Communication between the front-end and the trace information is done through Python, partly as Mercurial's web view theming engine is implemented in Python.

Scenting of the widgets is done through a combination of JavaScript and HTML5 <canvas> drawing. All the code meant to run client-side has been optimized where possible in order to allow for a smooth experience.

3 Sample Uses

ExecVus is not yet ready to tackle all duties related to debugging that we plan it to one day excel at, but a few tasks can be currently accomplished with the existing feature set.

For debugging, ExecVus can show developers what code is executed, although this feature is currently only available at the function-level. Identifying at a glance whether a function has been executed is important, as it reduces the time spent in debugging trying to trigger all the code paths needed to arrive to that function. Alternatively, the bug may be related to a certain function, though expected to run, being skipped.

Another task ExecVus can give quick information about is malware detection and understanding. Using scented widgets we can quickly present to developers an overview of the code paths that a certain function is involved in. A function that is only rarely involved in certain paths may indicate curious behaviour worth further analysis. Scented widgets let developers decide which functions are worth looking at by visually encoding the level of activity recorded.

4 Related Work

ExecVus would not be possible without Tralfamadore, as it is a visualization system for the data produced by the latter. We hope that the ideas proposed are extensible to other source-code visualization systems, whether analysis or development oriented.

SeeSoft [3] is a source code visualization system that displays line-oriented software statistics, such as code age. An important contribution of SeeSoft to Software Visualization is the full file display, with bars used to represent code lines. For SeeSoft's goal of presenting metrics on the whole of the code, this view is important as a way to show the full source file at once, while still allowing for lines to be distinguishable. We assessed the value of replicating SeeSoft's view for our file display, but further consideration of our task scenarios made it unnecessary. For trace analysis to be useful, the recorded information will be focused enough that we would not have annotations for every single line of code. As Tralfamadore aims to be a dynamic, query-based analysis tool, some of the data filtering will be done before ExecVus produces the visual representation.

Orso et al. [4] discuss various ways to visualize software at different levels of detail (system-, file- and statement-level). The system-level view is not something we are currently considering for ExecVus, although that may change in future iterations. We found the proposed file- and statement-level views to be lacking in a few ways, most importantly by encoding nested blocks both through colour and position (indentation). The colour encoding is of little benefit because it does not add execution information but simply reinforces the logical structure of the source code.

Path Projection [5] uses a simple code display, typical of IDEs, but pulls in the called functions at the location from which they are called, allowing the developer to see flow without navigating to another window. We think this view is very powerful and, given the tasks considered for ExecVus, we aim to implement a variation that accounts for multiple paths of interest being shown at once.

Scented widgets are an idea introduced by Heer et al. [6] that we felt had great applicability to our problem. There are many different locations in our system where scented widgets could have been used, but as an aid to navigation we felt they were most suited for function names. This way they serve their intended purpose of giving

[1] http://mercurial.selenic.com

[2] http://www.jquery.com

information about the result of an action even before the action is executed.

5 Evaluation

We consider the features currently available in ExecVus make a compelling argument for the tool and for future work. All choices made in developing the system were executed with developers and task scenarios in mind, and as such multiple attempts were made to minimize the effort required in learning the tool and its features.

5.1 Strengths

Even though ExecVus is not aimed at drop-in or casual users, we feel that the system is relatively easy to navigate for developers generally familiar with IDEs. The source view is central to our tool and a good carry-over from traditional development environments, easing users into their new role as execution analysts. The expanding view works similarly to how code folding works in IDEs that have the feature, and the animation adds functional and aesthetic value.

Scented widgets reinforce navigation cues by providing an idea about how interesting a function is. They allow the developer to decide, even before expanding the code view, which functions hold the most information.

Colour choices also help navigation by providing quick, highly recognizable markings for places of interest. Even a collapsed source file can span multiple screen-lengths, and a developer may quickly scroll through to see if annotations are available. With highly saturated colour bars attached to relevant lines of code, targets are visible even when fast-scrolling. The quiet black and gray used for general text, as well as the darker blue used to encode traced functions do not attract the eye as much as the colour bars. The positioning of the bars also makes them easier to spot, since they are not interspersed with the text.

We consider the rainbow colour scheme appropriate in this context, as there is no intended progression from red to blue (as some other tools using this colour scheme employ). We are simply interested in an effective way to maximize colour differences.

Since ExecVus is a web-based tool, extensions to the interface can be made without requiring a massive rewrite of the front-end or the need for a dedicated plugin interface. The learning curve for adding these extensions varies depending on the level of interaction with the backend that is required, which includes any meaningful work to be done with control-flow analysis. The data that is currently exposed to the front-end focuses on function calls exclusively.

5.2 Weaknesses

Some of ExecVus strengths are also causes of weakness, the most notable of which is colour. While the algorithm used to generate colours gives us a nice distribution that minimizes colour duplication, it does not ensure colour-blind safety, leaving certain users unable to use the full power of the system. This is especially true in combination with the way in which we choose 'seed' values for Hue, based on the line number, as certain seeds are particularly bad for colour blind users.

The reasoning behind choosing the seed value based on line numbering has been explained as a way to ensure that no false connections are made between bar charts using the same colours. The variation can be too subtle, as seen in Fig. 5, and it may not be perceived at all by some users, depending on the viewing display's quality and calibration, as well as the user's own optic ability.

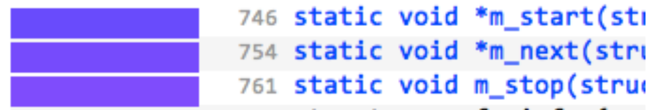

Figure 5. Subtle colour differences are hard to perceive, even though the functions they refer to are unrelated.

Even with a good algorithm to choose colours as 'far apart' from one another as possible, the problem of colour cacophony still presents itself when the scent has to work for functions that have upwards of 10 callers. This problem doubles when we consider the thickness of the individual bands also decreases to being nearly-invisible.

The choice of a standard size for the stacked bar graphs may mislead certain users into thinking that the aggregate number of calls is the same for each function.

Lastly, No visual cues are offered in directory listings (Fig. 6) as to where interesting information can be found, beyond a visual indication of whether annotations exist or not.

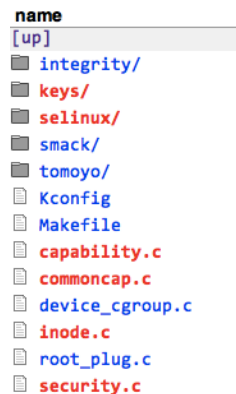

Figure 6. Directory Listing. Red denotes a file with annotations, or a folder that contains such files. No guidance available to help decide where to look next.

5.3 Lessons Learned

Interesting problems are more difficult than one might originally expect or calculate. The unforeseen issues that slowed down development of ExecVus were mostly related to the fact that Tralfamadore is also a tool under development, though with the advantage of a hefty head-start. The biggest difficulty was finding a data set that was fairly recent and complete, to allow all the manipulations required by this front-end tool. More time has been spent reverse-engineering the trace than doing anything else, and this cost was not expected at the onset.

While literature in the field mentions this repeatedly, colour choices are very difficult to make, especially for systems with a variable number of bins requiring distinct colours. Our encoding required multiple dimensions to be colour-coded (function-scope, file-scope), and consequently trade-offs were required to ensure adequate colouring can be achieved. Unfortunately, these trade-offs limit the tool's usability by the colour-blind and the sight-impaired.

It's fastest to learn a new tool or programming language when you need it the most.

6 Future Work

An important weakness of ExecVus is the lack of a front-end for inspecting the control-flow. Some backend code exists to accomplish this task, but lack of time prevented development of a suitable front-end. The planned implementation would allow the user to click a called function and 'zoom into' that function. A first pass would simply keep a list of links to previous locations as breadcrumb navigation, while a more advanced implementation could closely mimic the approach taken in Path Projection.

Other areas of improvement address the observed weaknesses of the system as follows:

For scent encoding it would be advisable, irrespective of colour choices, to encode the called function names in the view as well. A possible solution would be to present the name of the function as a hovering popup when the user points their mouse over the corresponding colour bar. In this manner we could also display call counts and other statistical information that is pertinent to the task (although care must be taken not to cause information overload). An extension of this point is the lack of a way to visualize where the most calls are made in any given function.

The size of the bar charts should be relative to the total count for the file to allow for effective comparison. It is also advisable to increase the size of the bar graphs so that individual bands are easily distinguishable. The portion of the page allotted to the code view can be safely decreased, as the more interesting task at this point is the identification of interesting locations in the code. While this may allow for more pixels available for drawing the different bands, we are still faced with the possibility of having to display very small bands (for functions involved in tens or hundreds of paths). One approach would be to limit the number of bands shown at any given time, and provide a visual cue that some have been omitted. A fisheye view can be used to magnify areas with thin bands.

Under-the-hood improvements may be made to move as much of the processing out of the client's browser and into the server backend. This can benefit from caching, where appropriate, or simply faster processing allowing for a smoother experience. JavaScript should not be used for complex string parsing or multiple array traversals.

7 Conclusion

We have begun development on ExecVus, a visualization system to be used in conjunction with the Tralfamadore execution mining system. The tool is aimed at developers looking to debug complex code, but that need to first understand the software's behaviour. It is also aimed at security researchers analyzing malware, by focusing on and making obvious the paths taken by code that is actually used in the course of execution.

Upon implementing some of the suggestions for future work, we believe ExecVus can stand on its own as a great first tool for those interested in understanding software behaviour.

References

[1] Jobs, Steve (August 2006). Live from WWDC 2006: Steve Jobs Keynote *http://www.engadget.com/2006/08/07/live-from-wwdc-2006-steve-jobs-keynote/* Retrieved 2009-12-16

[2] G. Lefebvre, B. Cully, M. J. Feeley, N. C. Hutchinson, A. Warfield. Tralfamadore: Unifying Source Code and Execution Experience *http://people.cs.ubc.ca/~andy/papers/tralf-eurosys-final.pdf* Retrieved 2009-12-16

[3] S.C. Eick, J.L. Steffen, E.E. Sumner Jr., Seesoft - A Tool for Visualizing Line Oriented Software Statistics" *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 957-968, November, 1992

[4] A. Orso, J. Jones, M.J. Harrold. Visualization of Program-Execution Data for Deployed Software *Proceedings of the ACM Symposium on Software Visualization*, San Diego, CA, USA, June 2003

[5] K.Y. Phang, J. S. Foster, M. Hicks, V. Sazawal. Path Projection for User-Centered Static Analysis Tools. *8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering,* Atlanta, GA, USA, November 2008.

[6] W. Willett, J. Heer, M. Agrawala. Scented widgets: Improving navigation cues with embedded visualizations. *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, iss. 6, pp.1129, 2007