Path Projection for User-Centered Static Analysis Tools

Khoo Yit Phang Jeffrey S. Foster Michael Hicks Vibha Sazawal

University of Maryland, College Park
{khooyp,jfoster,mwh,vibha}@cs.umd.edu

ABSTRACT

The research and industrial communities have made great strides in developing sophisticated defect detection tools based on static analysis. To date most of the work in this area has focused on developing novel static analysis *algorithms*, but has neglected study of other aspects of static analysis *tools*, particularly user interfaces. In this work, we present a novel user interface toolkit called Path Projection that helps users visualize, navigate, and understand program paths, a common component of many tools' error reports. We performed a controlled user study to measure the benefit of Path Projection in triaging error reports from Locksmith, a data race detection tool for C. We found that Path Projection improved participants' time to complete this task without affecting accuracy, while participants felt Path Projection was useful and strongly preferred it to a more standard viewer.

1. INTRODUCTION

Recent years have seen major advances in the development of defect detection tools based on static analysis. However, most research to date has focused on designing new static analysis *algorithms*. We believe it is equally important to study the other aspects of static analysis *tools*. Indeed, Pincus states that *"Actual analysis is only a small part of any program analysis tool [used at Microsoft]. In PREfix, [it is] less than 10% of the 'code mass'."* [23].

Generally speaking, static analysis tool users must perform two tasks: *triage*, deciding whether a report is a true or false positive, and *remediation*, fixing a true bug. An effective tool will assist the engineer in performing these tasks. However, while many tools support categorizing and managing error reports, most provide limited assistance for deciding whether a report is true, and if so, how to fix it.

To address this problem, we present *Path Projection*, a new user interface toolkit that helps users visualize, navigate, and understand *program paths* (e.g., call stacks, control flow paths, or data flow paths), a common component of many static analysis tools' error reports. Path Projection aims to help engineers understand error reports, improving the speed and accuracy of triage and remediation.

Our toolkit accepts an XML error report containing a set of paths and automatically generates a concise source code visualization of these paths. First, we use *function call inlining* to insert the bodies of called functions just below the corresponding call sites, rearranging the source code in path order. Second, we use *code folding* to hide potentially-irrelevant statements that are not involved in the path. Finally, we show multiple paths *side by side* for easy comparison. By synthesizing the visualization directly from the error report, Path Projection greatly reduces the user's effort to examine the code for the error report.

Additionally, we have observed that users often lack sufficient experience or knowledge of a particular static analysis tool to triage its error report efficiently. Instead, users will often spend time discerning information about the program that is not relevant to the triaging task. Our solution to this problem is a tool-specific *checklist* that gives users a systematic procedure to perform triage. Anecdotal observations from our pilot studies suggest that checklists can dramatically reduce triaging times. We believe checklists can be useful additions to many static analysis interfaces, and merit further study.

We have evaluated Path Projection's utility by performing a controlled experiment in which users triaged reports produced by Locksmith [24], a static data race detection tool for C. We measured users' completion time and accuracy in triaging Locksmith reports, comparing Path Projection to a "standard" viewer that we designed to include the textual error report along with commonly-used IDE features. Both interfaces also include a checklist for Locksmith to keep our participants focused on the triaging task.

Our results show that Path Projection reduced users' times to triage an error report by roughly one minute, an 18% improvement, without reducing accuracy. Also, seven of eight users preferred Path Projection over the standard viewer.

In summary, this paper makes two main contributions. First, we present *Path Projection*, a novel toolkit for visualizing program paths (Section 3). While mature static analysis tools can have sophisticated graphical user interfaces (Section 7), these are often integrated and cannot easily be used by other tools. In contrast, we show how to apply Path Projection to Locksmith and to BLAST [5].

Second, we present quantitative and qualitative evidence of Path Projection's benefits in triaging Locksmith error reports (Sections 5 and 6). To our knowledge, ours is the first study to consider the task of triaging defect detection tool error reports, and the first to consider the user interface in this context. Our study results provide some scientific un-

derstanding of which features are most important for making users more effective when using static analysis tools.

Due to space constraints, our discussion of Path Projection, checklists, and our experiments is brief. Further details can be found in a companion technical report [18].

2. MOTIVATION: PROGRAM PATHS

The simplest way for a static analysis tool to report a potential defect is to indicate a line in the file where the defect was detected. However, while this works reasonably well for C or Java compiler errors, static analysis designers have long realized it is insufficient for understanding the results of more sophisticated static analyses. Accordingly, many static analysis tools provide a *program path*, i.e., some set of program statements, with each error message. For example, CQual [15] and MrSpidey [11] report paths corresponding to data flow; BLAST [5] and SDV [3] provide counterexample traces; Code Sonar [14] provides a failing path with pre- and post-conditions; and Fortify SCA [12] provides control flow paths that could induce a failure.

Because static analysis tools often make conservative assumptions, they typically produce *false positives*, i.e., reports that do not correspond to actual bugs. The user must therefore *triage* a tool's reports by, e.g., tracing the reported paths, to decide if a problem could actually occur at runtime.

Unrealizable paths in Locksmith. To understand the challenges that occur when tracing program paths, we consider the problem of triaging error reports in Locksmith, a data race detection tool for C [24]. Locksmith reports paths whose execution could lead concurrent threads to access a shared variable simultaneously. However, like many other tools, Locksmith employs a *path-insensitive* analysis, meaning it assumes that any branch in the program could be taken both ways. Thus, to triage an error report, a user must decide whether a pair of reported accesses is *simultaneously realizable*, i.e., if there is some execution in which both could happen at once.

The triaging process is conceptually simple: we must examine the control flow logic along the paths and ensure it does not prevent both accesses from occurring at once. However, in practice, performing this task is non-trivial, taking a surprising amount of effort using typical code editors.

Consider Figure 1, a screenshot of our *standard viewer*, which represents the assistance a typical editor or IDE would give users in understanding textual error reports. A sample Locksmith error report is shown in the pane labeled (1).[1] This error report comes from aget, a multithreaded FTP client. The report first identifies prev as the shared variable involved in the race. Then, it lists two call stacks leading to conflicting accesses to prev, the first via a thread created in main, and the second via a different thread created in resume_get. No lock is held at either access.

We arrived at the screenshot in Figure 1 by tracing through the error report. We began by clicking a hyperlink (2) to jump to the thread creation site for path 1. Next, we examined the code above the thread creation site, which required some scrolling, and saw that this thread is unconditionally created. Later, we will want to relate this thread creation site with the one in path 2, so we split the window to keep this code visible (3). Then, we inspected the function signal_waiter by clicking on the hyperlink (first selecting the lower viewer so the target of the link was shown there). We eventually created several splits to keep track of the relevant code of the remaining functions in the report.

The resulting display is cluttered and hard to manage, and if we were to continue, we would likely be forced to collapse splits, which would make it harder to refer back to code along the path. In general, when examining program paths with standard viewers, mundane tasks such as searching, scrolling, navigating, viewing, or combining occur with such frequency that they add up to a significant cognitive burden on the user and distract from the actual program understanding task. In our experience, it can be quite tedious to triage error reports even for small programs such as aget, which has only 2,000 lines of code. Since static analysis tools can yield hundreds of defect reports on large programs, we believe it is crucial to make triaging error reports easier.

The goal of Path Projection is to make tracing paths much easier by reducing the cognitive load on the user. In our user study, we focused on the task of triaging Locksmith error reports by looking for unrealizable paths. However, we also believe that Path Projection is generally applicable to many other program understanding tasks involving path tracing.

We should add that, besides unrealizable paths, there are several other reasons Locksmith may report a false positive, e.g., it may incorrectly decide that a variable is shared, or it may decide a lock is not held when it is [24]. For many programs that we have looked at, Locksmith's lock state analysis is accurate, and warnings often involve only simple sharing (e.g., of global variables, or via simple aliasing), so our current work focuses on tracing program paths. These other cases are certainly important, and we intend to address them in future work.

3. PATH PROJECTION

The input to our Path Projection toolkit is an XML error report and the program source. Path Projection combines two techniques—*function call inlining* and *path-derived code folding*—to "project" the source code onto the error path.

Figure 2 shows the Path Projection interface for the error report from Figure 1. Path Projection first lays out the code by inlining function calls (1), i.e., by inserting the code for each call along the path directly below the calling line. In the close-up of Figure 2, main calls resume_get, so the body of resume_get is displayed directly below the call site. Then resume_get calls http_get (via pthread_create), so the latter's body is inlined, and so on. As a result, the code is visually arranged in path order, so the user no longer has to jump around the program to trace a path.

Next, Path Projection applies code folding to hide away irrelevant code (2), so that the user is initially shown as much of the path as will fit in one screen. We show only lines that are implicated in the error report, and the function names or conditional guards of enclosing lexical blocks (including matching braces). This degree-of-interest model is similar to that proposed by Furnas [13]. Note that Path Projection's code folding is more selective than the block-level folding that is more common in code editors; in the latter, an entire lexical block has to be unfolded to reveal even a single implicated line. Our code folding heuristic can also be thought of as a simple form of program slicing [30].

Path Projection can also show paths *side by side*, with

[1]Note that this report format is slightly different than Locksmith's current output, but the differences are merely syntactic.

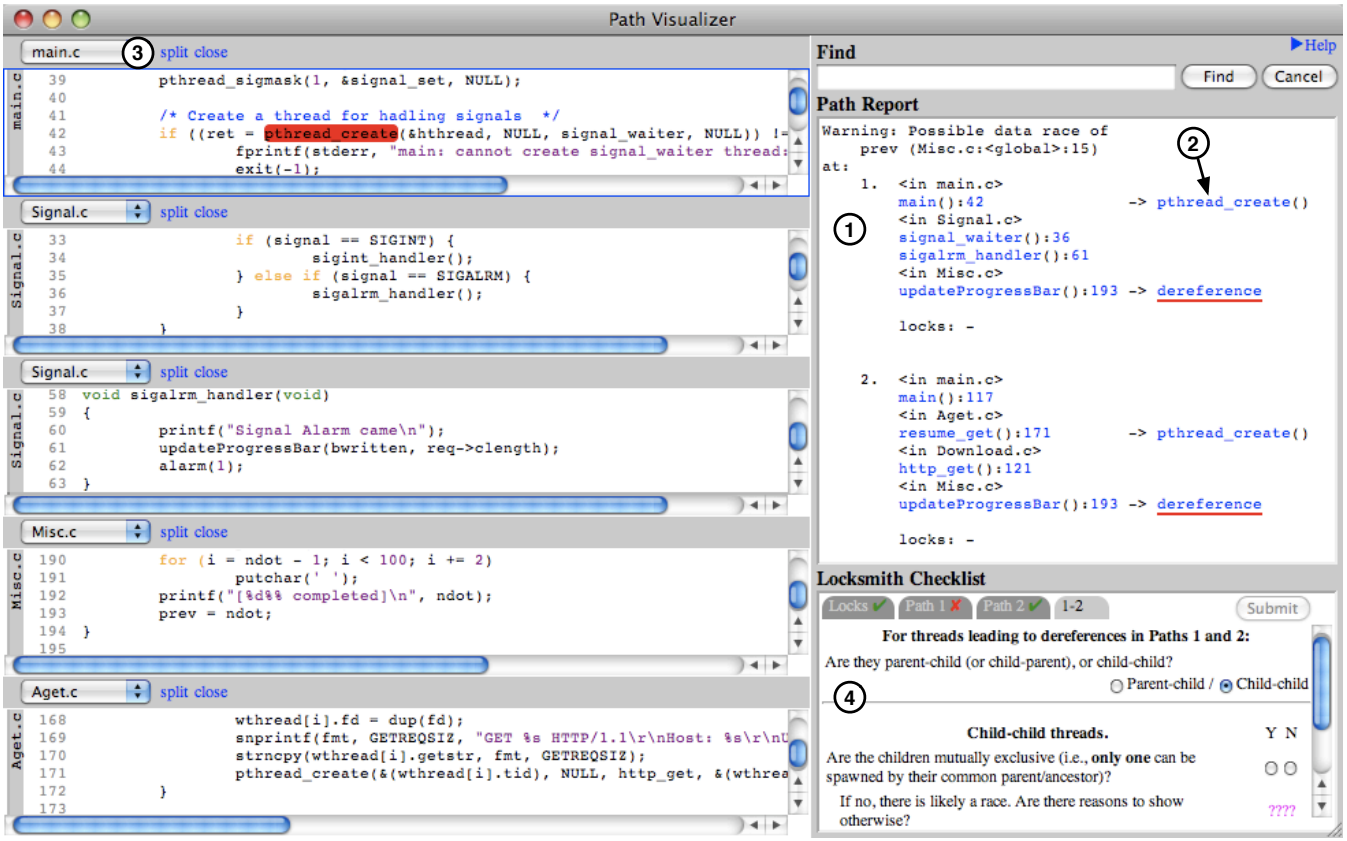

Figure 1: Standard viewer showing 5 splits: one for each function call in path 1, and the first function call in path 2 (picture in color; labels described in text)

inlining and code folding applied to each path separately to avoid conflating the context of different paths. Side-by-side path display is particularly useful for triaging Locksmith error reports, so the user may easily compare several paths.

At times during triaging, a user may wish to look at code outside a reported path. At a coarse level, the user may disable code folding for an entire function by clicking the corresponding expansion button. For finer control, the user may also use the *multi-query* feature to search for and highlight multiple terms at once (3). Matching terms are revealed despite folded code, and each term is given a distinct color to make it easy to locate. For Locksmith, we find multi-query especially useful when preset with pthread-related functions. Lastly, Path Projection includes a *reveal definition* facility that uses inlining to show the definition of any function or variable along the path.

Let us continue our earlier example, now using Path Projection to trace path 2. Unlike before, we can mostly ignore the error report, since it is apparent that main calls resume_get, which calls pthread_create in a loop, and so on. We can tell at a glance that the conflicting access is in updateProgressBar, underlined in red. Thus, we just need to examine the conditionals along the path, as we would with the other interface. If we are unfamiliar with the code, we can use reveal definition to show the body of read_log, used in the conditional in main, and multi-query to show uses of foffset in http_get. From this we can decide that the path is indeed realizable. Furthermore, we see there is another call to pthread_create earlier in main. This is the call in path 1, and so we now see that both paths may occur at once.

Notice that with Path Projection, we are no longer distracted by mundane tasks such as window management, and can focus on the task of understanding program paths. In many cases, this task becomes a simple matter of reading each column in the Path Projection display from top to bottom, and we can triage error reports more easily.

Applying Path Projection to other tools. Path Projection is designed to be a general toolkit for visualizing path-based error reports. More precisely, Path Projection is a standalone tool that takes as input an XML error report and the source code under analysis, and produces a web browser-based visualization of the code, as shown. The XML format encodes paths as lists of function calls, function returns, and any implicated lines in those functions.

In addition to Locksmith, we have applied Path Projection to counterexample traces produced by BLAST [5], a software model checking tool. Such traces are not call stacks, as in Locksmith, but rather are execution traces that include function calls and returns. We use a short awk script to post-process a BLAST trace into our XML format.

We believe that Path Projection makes it much easier to understand BLAST's error reports. BLAST error reports can be confusing in their textual form. For example, paths can be hundreds of lines long and, as a result, it can be quite difficult to keep track of the execution context, e.g., by matching function calls and returns. Using Path Projection for BLAST reports, however, it is quite easy to tell when functions are called, and when they return. In fact, the structure of the path is usually apparent without having to

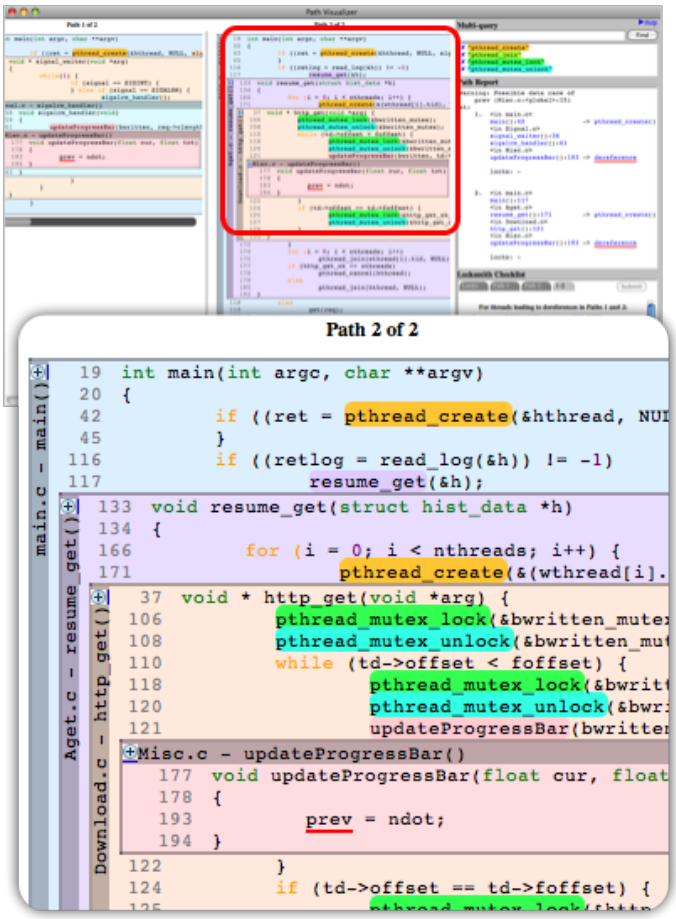

Figure 2: Path Projection (top) and a close-up. This and additional screenshots are available at `http://www.cs.umd.edu/projects/PL/PP`.

refer to the error report. Our technical report [18] contains a screenshot and more detailed discussion of using BLAST with Path Projection.

4. CHECKLIST

To understand an error report, the user must develop a way to take the information in the report and relate it to the potential problem with the code. Moreover, to decide whether an error report is a false positive, the user has to understand something about the sources of imprecision in the analysis. To static analysis experts, this *triaging procedure* often seems obvious, but participants in our pilot studies (who were not static analysis experts) had trouble even with extensive tutorials. Participants often developed their own ad hoc, inconsistent procedures that neglected some sources of imprecision (and thus sometimes wrongly concluded a report to be a true bug) or assumed non-existent sources of imprecision (and therefore wasted time verifying conditions certain to hold).

We addressed this situation by encoding the triaging procedure as a *checklist* that enumerates the steps required to triage a specific report. A triaging checklist is tool- and error-specific: different tools have different sources of imprecision and different kinds of error reports.

For Locksmith, the principle source of imprecision we are concerned with is unrealizable paths, and the error reports list a set of program paths that access a shared variable with inconsistent locking. Thus, our checklist for Locksmith breaks down triaging into a set of sub-tasks: checking simultaneous realizability for each pair of possibly-racing paths listed in the error report.

Figure 1 (4) shows part of our checklist, a tab that asks the user to decide whether paths 1 and 2 may race. These tabs are automatically generated from the error report. As checklists are not user-interface specific, we included the same checklist in both the Path Projection and Standard Viewer.

This particular tab first directs the user to determine whether the two threads are in a parent-child or other (child-child) relationship. If the user selects child-child, as shown, then the user has to check whether the two children are mutually exclusive, e.g., spawned in mutually exclusive branches of an if statement, which would preclude a race. If they are not, the last question simply asks the user if there is a race for reasons that we did not anticipate. Details of other tabs can be found in our companion technical report [18].

Comparing the results of our pilot study to the one reported in the next section, users triaged error reports 41% faster with checklists than without them. Though there were other differences in the procedures and the interfaces, our observations suggest that most of the improvement is due to the checklist. Therefore, we believe that a tool-specific checklist has independent value, and that tool developers should consider designing checklists for use with their tools.

5. EXPERIMENTAL EVALUATION

We evaluated Path Projection's utility in a controlled user study in which participants triage Locksmith error reports using either Path Projection (PP) or the standard viewer (SV). In this task, PP provides an advantage over SV in triaging if it is faster, easier, and/or more accurate. The discussion of our experiments and results is brief due to space constraints. Full details can be found in our companion technical report [18].

Participants. We recruited eight student participants (3 undergraduate, 5 graduate) from the UMD Computer Science Department. All participants had taken at least one college-level class involving multithreaded programming (not necessarily in C). On a 1 (no experience) to 5 (very experienced) scale, participants rated themselves as 3 or 4 in their ability to debug data races. Two participants had previously used a race detection tool (Locksmith and Eraser [27]).

Procedure. Each participant performs the triaging task in two sessions, first with one interface and then with the other. All participants receive the same set of problems in the same sessions: 1.1, 1.2, and 1.3 in Session 1, and 2.1, 2.2, and 2.3 in Session 2. However, half of the participants use PP in Session 1 and half use SV, and the participants switch interfaces for the second session.

Each session begins with short tutorials on pthreads and data races in general, and on Locksmith in particular, with emphasis on the items in the triaging checklist. The particular features of the user interface (PP or SV) are explained in a subsequent tutorial.

Next comes a single practice trial and three actual trials, all of which follow the same procedure. In each trial, we first ask participants to triage a Locksmith error report. Triaging ends when participants complete and submit the triaging

Completion times and accuracy for each trial

	Session 1				Session 2			
Trial	1.1	1.2	1.3	mean	2.1	2.2	2.3	mean
User	SV				PP			
1	8:36	14:14	9:44	10:51	7:07	4:48*	4:02	5:19
2	5:07*	3:10	5:50	4:42	4:16	2:29*	2:10*	2:58
5	7:46	2:34	5:38*	5:20	5:13	3:43*	1:18*	3:25
7	5:40	6:23	7:35	6:33	3:05	3:53*	2:32	3:10
				6:51				**3:43**
User	PP				SV			
0	6:27*	6:09	8:32*	7:03	9:42	5:16*	3:11*	6:03
3	6:38	7:18	8:35	7:30	11:18	6:21	3:39	7:06
4	8:21	2:11	4:43	5:05	5:26*	4:27*	2:46*	4:13
6	7:11	2:52	4:50**	4:57	4:33	4:06*	1:58*	3:32
				6:09				**5:14**
# Tabs	3	2	6		6	3	3	

* one incorrectly answered tab in the checklist

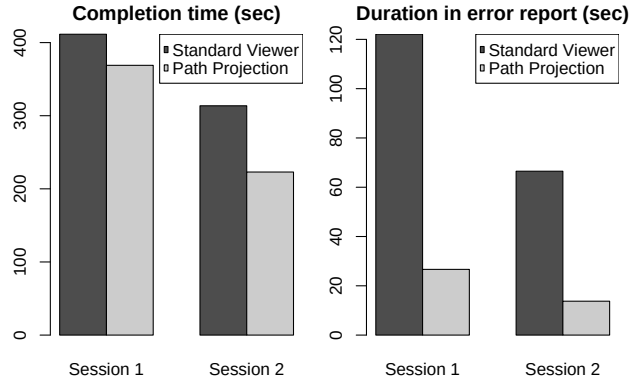

Figure 3: Quantitative data

checklist. Immediately afterward, we present participants with the same problem and ask them to explain out loud the steps they took to verify the warning. This allows us to compare their descriptions with our expectations, and gives the users a chance to revisit their reasoning without bias from the experimenter. After the experiment, participants complete a questionnaire and interview to determine their opinion of the user interface.

We ran the experiment on Mac OS X 10.5.2 with a 24-inch wide-screen LCD monitor. To avoid bias due to OS competency, all keyboard shortcuts were disabled except for cut, copy, paste, find, and find-next, and participants were informed of these shortcuts.

Programs. Each trial's error report was drawn from a program to which we had previously applied Locksmith [24]. Reports from programs engine and pfscan were used during the tutorial and practice trials. Trials 1.1–1.3 use reports from aget, and trials 2.1–2.3 use reports from knot. Overall, there were 23 checklist tabs to complete for the experiment, 8 true positives and 15 false positives. Note that we chose reports in which imprecision in the sharing and lock state analysis do not contribute to the report's validity, so as to focus on the task of tracing the program path.

6. EXPERIMENTAL RESULTS

Completion time. We measured the time it took each participant to complete each trial, defined as the interval from loading the user interface until the participant submitted a completed checklist. The top part of Figure 3 lists the results, and the chart in the lower-left corner of the figure shows the mean completion times for each interface/session combination. We found in general that PP results in shorter completion times than SV.

More precisely, the mean completion time is 6:02 minutes for SV and 4:56 minutes for PP, a 1:06 minute (or 18%) reduction. A standard way to test the significance of this result is to run a user interface (within subjects) by presentation order (between subjects) mixed ANOVA on the mean of the three completion times for each participant in each session. This test revealed a statistically significant interaction effect between the interface and the presentation order ($F(1,6) = 6.046, p = 0.004$).[2] We believe this is a learning effect—notice that for the SV-PP order (SV first, PP second), the mean time improved 3:08 minutes from the first session to the second, and for the PP-SV order the mean time improved 55 seconds. We ran two one-way, within-subjects ANOVAs, analyzing each presentation order separately, and found that both of these improvements were statistically significant. ($F(1,3) = 12.78$, $p = 0.038$ for SV-PP, and $F(1,3) = 19.33$, $p = 0.022$ for PP-SV).

However, notice that the SV-PP improvement is much greater than the PP-SV improvement. We applied Cohen's d, a standard mean difference test, which showed that the SV-PP improvement was large ($d = 1.276$), and the PP-SV improvement was small-to-medium ($d = 0.375$). This provides evidence that while there is a learning effect, PP still improves user performance significantly.

Note we did not distinguish correct and incorrect answers when analyzing completion times. It is unclear how incorrect answers affect timings, especially since in many cases, only one tab out of several contained an incorrect answer.

Accuracy. The chart at the top of Figure 3 also indicates, for each trial, how many checklist tabs were answered incorrectly. The total number of checklist tabs to complete per trial is at the bottom of the chart. Our results show that user mistakes were evenly distributed across both interfaces, 10 (10.9%) under PP and 9 (9.8%) under SV. For each participant, we summed the number of mistakes in each session and compared the sums for each interface using a Wilcoxon rank-sum test. This showed that the difference is not significant ($p = 0.770$), suggesting the distribution of errors is comparable for both interfaces. Thus, using PP, participants came to similarly accurate conclusions in less time.

Time in error report. The chart in the bottom right of Figure 3 shows the mean times spent with the mouse hovering over the error report for each session and interface. On average, participants spent 20 seconds in the error report under PP, compared to 94 seconds under SV, a dramatic difference. We believe this difference occurs because PP makes paths clearly visible in the source code display, whereas in SV, hyperlinks in the error report are heavily used to reveal relevant parts of the code. One participant even noted that the error report is "necessary for the standard viewer, but just a convenience in [PP]."

Qualitative results. We asked each participant to rate, on a scale of 1-5, whether each interface was *quick to learn*, led them to *confidence in their answer*, and made *races easy to verify*. There was no statistically significant difference in users' responses between the two interfaces. However,

[2] As is standard, we consider a p-value of less than 0.05 to indicate a statistically significant result.

when asked whether they preferred PP overall, 7 out of 8 responded that they somewhat or strongly agreed.

We also asked users to rate individual features of PP on a scale of 1-5, in particular the *error report*, *checklist*, *function inlining*, *code folding*, *multi-query*, and that *multi-query reveals folded code*. Participants rated all these features somewhat or very useful, with statistical significance.

The checklist was very well received overall. One participant said, "[It] saved me from having to memorize rules." Interestingly, two participants felt that, while the checklist reduced mistakes, it made triaging take longer. This conflicts with our experience—as we mentioned earlier, participants in our pilot study, which did not include the checklist, took notably longer to triage error reports than participants in our current study.

We thought that participants would be wary about function inlining and code folding, since the former is an unfamiliar visualization and the latter hides away much code. However, participants rated both very highly, saying particularly that code folding was "the best feature," "my favorite feature," and "I love using this feature [code folding]."

Threats to validity. There are a number of potential threats to the validity of our results. The core threat is whether our results generalize. We had a small number of participants, all of whom were students rather than expert programmers. The set of programs ($N = 2$) and error reports ($N = 6$) in the experiment was small. Moreover, participants were asked to triage error reports for unfamiliar programs. However, even with these limitations, our experiment did produce statistically significant and anecdotally useful information for our population and sample trials. We leave carrying out a wider range of experiments to future work.

One minor threat is that the SV interface is not a production-quality IDE. This was a deliberate choice to reduce bias from prior IDE experience, but we may have omitted features that would be useful for our task.

Another concern is whether the checklist could bias the results. We do not think so, since the checklist is derived from the imprecision and error report format of Locksmith, as discussed in Section 4, rather than being interface-specific.

7. RELATED WORK

Error report triaging is essentially a program comprehension task. There has been substantial research into tools for assisting program comprehension tasks, often related to code maintenance and re-engineering (e.g., SHriMP [29] and Code Surfer [1], to name just two recent tools). As far as we are aware, none of these tools has been specifically designed to support users of static analysis for defect detection.

Conversely, many defect detection tools that use static analysis provide custom user interfaces [3, 21, 7, 17, 14, 1, 11, 6], IDE plug-ins [19, 12, 28, 15], or both [25]. However, we are not aware of any prior studies that measure the impact of the user interface on a static analysis tool's efficacy. Indeed, many proprietary tools have licenses that prohibit publishing the results of studies about their tools.

We surveyed these tools based on publicly-available screenshots and literature and compared their features with Path Projection (Table 1). Most interfaces share some features with Path Projection, but Path Projection's multi-query and path-derived (error-specific) code folding seem to be new; no tool provides a checklist to help with triaging.

	Features (legend below)						
Tool	HLR	HPC	CF	FCI	AC	AQ	OV
Path Projection	×	×	×	×			
SDV [3]	×				×		
PREFix [21]		×			×		
PREFast [21]		×		*n/a*	×		
Prevent [6]	×				×		
Code Sonar [14]	×	×	×	×	×		
MrSpidey [11]	×	×				×	×
Fortify SCA [12]	×		×				×
Klocwork [19]	×		×				
Fluid [28]	×		×				
FindBugs (GUI) [25]	×	×		*n/a*			
FindBugs (Eclipse)	×		×	*n/a*			
CQual [15]	×						

HLR : hyperlinked error report
HPC : highlighted path in code
CF : code folding
FCI : function call inlining
AC : annotated code with analysis
AQ : analysis queries
OV : other visualization (see text)

Table 1: Summary of tool interface features

Code Sonar has many features in common with Path Projection, but we believe its design could be improved. For example, several of Code Sonar's features interleave colored lines of text with the source code, causing the source code to be discontinuous and hard to follow. A few tools annotate the source code with summaries of their analysis, a feature not (yet) in Path Projection. MrSpidey also allows the user to interactively query its analysis for possible run-time values of a given expression. Two tools provide graphical visualization of paths: Fortify SCA illustrates paths as UML-style interaction diagrams, and MrSpidey overlays lines on the source code to illustrate value flows.

Research in bug triaging has focused on detecting duplicate bug reports [26], assigning bug-fixing responsibility [2, 8], or visualizing the progress of bug reports [9]. There has also been work on improving error messages [10, 22, 4, 16, 31], and on organizing them by priority [20]. Path Projection, and our study, differs from this work in exploring how triage can be performed more accurately and efficiently.

8. CONCLUSIONS

This paper introduced Path Projection, a new program visualization toolkit designed to help programmers navigate and understand program paths, a common output of many static analysis tools. We measured the performance of Path Projection for triaging error reports from Locksmith and found that, compared to a standard viewer, Path Projection reduces completion times with no adverse effect on accuracy.

We believe our work is the first to study the impact of a user interface on the efficiency and accuracy of triaging static analysis error reports. In future work, we intend to explore other potential applications of Path Projection beyond bug triage, and to study the use of checklists to complement static analysis tools. Path Projection is available at `http://www.cs.umd.edu/projects/PL/PP`.

Acknowledgments

We thank François Guimbretière, Bill Pugh, Chris Hayden, Iulian Neamtiu, Brian Corcoran, and Polyvios Pratikakis, and our study participants for their help with this research. This research was supported in part by National Science Foundation grants IIS-0613601, CCF-0430118, and CCF-0541036.

9. REFERENCES

[1] P. Anderson, T. Reps, T. Teitelbaum, and M. Zarins. Tool support for fine-grained software inspection. *IEEE Software*, 20(4):42–50, July/August 2003.

[2] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *ICSE '06*, pages 361–370, 2006.

[3] T. Ball and S. K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *POPL'02*, pages 1–3, 2002.

[4] M. Beaven and R. Stansifer. Explaining type errors in polymorphic languages. *ACM Lett. Program. Lang. Syst.*, 2(1-4):17–30, 1993.

[5] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. The Blast query language for software verification. In *SAS'04*, pages 2–18, 2004.

[6] Coverity, Inc. Coverity Prevent SQS, 2007. `http://www.coverity.com/html/prod_prevent.html`.

[7] R. F. Crew. ASTLOG: A Language for Examining Abstract Syntax Trees. In *Proceedings of the Conference on Domain-Specific Languages*, Santa Barbara, California, Oct. 1997.

[8] D. Cubranic and G. C. Murphy. Automatic bug triage using text categorization. In *SEKE'04*, pages 92–97, 2004.

[9] M. D'Ambros, M. Lanza, and M. Pinzger. "A Bug's Life" Visualizing a Bug Database. *VISSOFT '07*, pages 113–120, 24-25 June 2007.

[10] D. Duggan and F. Bent. Explaining type inference. *Science of Computer Programming*, 27(1):37–83, 1996.

[11] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Catching Bugs in the Web of Program Invariants. In *PLDI '96*, pages 23–32, 1996.

[12] Fortify Software Inc. Fortify Source Code Analysis, 2007. `http://www.fortifysoftware.com/products/sca/`.

[13] G. W. Furnas. Generalized Fisheye Views. In *CHI'86*, pages 16–23, 1986.

[14] GrammaTech, Inc. CodeSonar, 2007. `http://www.grammatech.com/products/codesonar/overview.html`.

[15] D. Greenfieldboyce and J. S. Foster. Visualizing Type Qualifier Inference with Eclipse. In *Workshop on Eclipse Technology eXchange*, Vancouver, British Columbia, Canada, Oct. 2004.

[16] C. Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Sci. Comput. Program.*, 50(1-3):189–224, 2004.

[17] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A System and Language for Building System-Specific, Static Analyses. In *PLDI '02*, pages 69–82, 2002.

[18] Y. P. Khoo, J. S. Foster, M. Hicks, and V. Sazawal. Path Projection for User-Centered Static Analysis Tools. Technical Report CS-TR-4919, Department of Computer Science, University of Maryland, College Park, Aug. 2008.

[19] Klocwork Inc. Klocwork Enterprise Development Suite, 2007. `http://www.klocwork.com`.

[20] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler. Correlation exploitation in error ranking. In *FSE '04*, pages 83–93, 2004.

[21] J. R. Larus, T. Ball, M. Das, R. DeLine, M. Fahndrich, J. Pincus, S. K. Rajamani, and R. Venkatapathy. Righting Software. *IEEE Software*, 21(3):92–100, May/June 2004.

[22] M. Neubauer and P. Thiemann. Discriminative sum types locate the source of type errors. In *ICFP '03*, pages 15–26, 2003.

[23] J. Pincus. User Interaction Issues in Defect Detection Tools. Presentation at UW/MSR Research Summer Institute, 2001. `http://research.microsoft.com/users/jpincus/uwmsrsi01.ppt`.

[24] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: Context-Sensitive Correlation Analysis for Race Detection. In *PLDI '06*, pages 320–331, 2006.

[25] B. Pugh et al. FindBugs, 2007. `http://findbugs.sourceforge.net`.

[26] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of Duplicate Defect Reports Using Natural Language Processing. In *ICSE '07*, pages 499–510, 2007.

[27] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. In *SOSP '97*, pages 27–37, 1997.

[28] W. Scherlis et al. The Fluid Project, 2007. `http://www.fluid.cs.cmu.edu:8080/Fluid`.

[29] M.-A. Storey. *A Cognitive Framework For Describing and Evaluating Software Exploration Tools*. PhD thesis, Computing Science, Simon Fraser University, Canada, 1998.

[30] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[31] J. Yang, J. Wells, P. Trinder, and G. Michaelson. Improved type error reporting. In *Workshop on Implementatino of Fnuctional Languages*, pages 71–86, 2000.