Interactive Binary Instrumentation

Chadd C. Williams
Department of Computer Science
University of Maryland
chadd@cs.umd.edu

Jeffrey K. Hollingsworth
Department of Computer Science
University of Maryland
hollings@cs.umd.edu

Abstract

In this paper we present a new model for instrumenting a process on the fly. We describe a command line tool developed by our research group based on our instrumentation library, Dyninst. With this model the user no longer needs to have a full set of mutations planned out ahead of time. The command line interface allows the user to mutate a process, gather data, and generate new mutations while the mutatee is running. This model effectively allows the user to put off most of the decisions about what mutations to insert into a mutatee until run-time, and allows the user to take a more active role in analyzing the data that comes back from the mutatee.

1. Introduction

The normal cycle of developing a program is to edit source code, compile it, and then execute the resulting binary. However, sometimes this cycle can be too restrictive. We may wish to change the program while it is executing, and not have to re-compile, re-link, or even re-execute the program to change the binary. At first thought, this may seem like a bizarre goal, however there are several practical reasons we may wish to have such a system. Runtime code changes, through dynamic instrumentation, are useful to support a variety of applications including debugging, performance monitoring, and application steering. Another key to this technique is that access to the source code of the program to be instrumented is not needed.

The Dyninst library provides users with a platform independent API to permit the insertion of code into a running program [2,6]. The traditional usage of this library has been for the user to write a mutator program that will attach to a running process, a mutatee, and insert and remove instrumentation code as necessary. The user needs to change only the mutator to change the instrumentation code and rerun the mutatee/mutator pair. However, this model still has shortcomings. The space of possible mutations and how they can be applied must be decided when the mutator is built. The user can write a shared library containing code to be run from instrumentation and load it into the mutatee's address space using Dyninst, but how that instrumentation is applied is still a build time decision.

In order to push decisions about instrumentation off completely until runtime, the user would have to explicitly make the mutator interactive to allow the user to guide the instrumentation by hand. Even then the user is limited to the instrumentation code available in the mutator.

2. DyninstAPI

The result of inserting instrumentation code is shown in Figure 1. To instrument a point in a program, an instruction in the text section is replaced with a jump to a *base trampoline*. The base trampoline contains a jump to a *mini-trampoline* and space for the replaced instruction. The mini-trampoline contains the instructions that perform the desired functionality as well as saving and restoring registers. The mini-trampoline can call an existing function or run a snippet of code created by Dyninst and inserted into the running program. The base and mini trampoline architecture allows many different bits of instrumentation (many mini-trampolines) to be inserted at the same point in the program, at different times. The overhead of executing instrumentation is similar to a function call and depends on how efficiently registers can be saved and restored during execution.

3. Command Line Instrumentation

The model discussed above is such that the user builds a mutator that contains a set of mutations that may be conditionally inserted into the mutatee. While this model has great advantages over recompiling the mutatee to change the instrumentation, it does not fit all needs. In the early phase of a project, the user may not know what to expect from the instrumentation. The data gathered from the mutatee may need to be studied before more instrumentation is written. The previous model provides no easy way to study the new mutatee behavior and produce new instrumentation without restarting the mutatee/mutator pair after updating the mutator. Once the mutator is compiled the possible set of mutations that can be inserted, and how they are inserted, is locked in. This is acceptable for short running mutatees or programs that do not require a long “warm-up” period. Mutatees that take even 30 minutes to warm up could present a problem in the early phase of the project as the user tries to

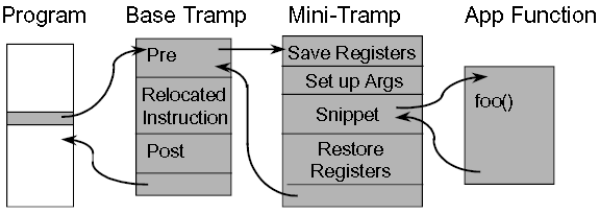

Figure 1: Detailed View of Instrumentation Code

understand exactly what type of instrumentation is needed.

Instrumenting code from the command line as the mutatee runs is a more effective model for the user does not want to restart often. Allowing the user to look at the data that comes back from the mutatee and then decide what type of mutations need to be inserted can be very helpful. The user can then take a more active role in analyzing the data returned from the mutatee and more easily deal with new and unexplored situations. It could very well be the case that the user may want to use this model the first few times mutations are inserted into a mutatee, to get a feel for the behavior of the mutatee and determine what data needs to be collected and how, before compiling the mutations into the more rigid form of a mutator. Just as the user does not want to recompile an application to insert instrumentation, the user may not want to restart an application to insert fresh instrumentation.

4. Dyner

Our research group has produced Dyner, a TCL-based interactive command line tool, to give the user a command line interface for instrumenting a program [5]. This allows the user to apply the model described above to instrument code. Dyner provides full access to Dyninst functionality and has a few macros defined for common tasks, such as function tracing or counting function executions for performance measurement. From the command line the user can declare variables in the mutatee, load shared libraries in the mutatee, instrument functions and save the mutated binary back to a runnable executable file on disk. Dyner also provides the user with the functionality necessary to attach to an already running process or to launch a process from an executable to instrument.

Dyner provides functionality to instrument various predefined points in a function, the entry and exit points, as well as to instrument a particular line in a source file. Breakpoints inserted with Dyner can be predicated by logic written at the command line in C like code. Dyner parses the input statements and then generates code that is inserted into the mutatee. Local and global variables, and functions declared in the mutatee can be used in the statements. Moreover, variables declared with Dyner can be included in the statements.

Dyner has some exploratory functionality as well. This allows the user to inspect the running process before doing any instrumentation. These functions

```
% declare int foo_counter;

% break foo exit { foo_counter>5; }

% at foo entry { foo_counter ++; }
```

Figure 2 Dyner Commands

allow the user to interactively learn about an unfamiliar executable. The *show* command in particular, can be used to retrieve a list of variables, functions or modules from the process, with options to restrict the command to a particular function or module. The *whatis* command allows the user to retrieve detailed information about functions or variables in the mutatee. For variables, the retrieved information includes the type, scope, line number and frame. The *find function* command will display type information regarding a function in the mutatee.

Dyner also support batch script files and includes a *source* command to run them. The user can write scripts of Dyner commands and run them from within Dyner. This allows the user to easily and reliably repeat common tasks.

Dyner also allows the user to interrupt the mutatee at any time. Once the mutatee arrives at a spot deemed safe by Dyninst it is paused and the user can modify instrumentation, inspect variables or load a shared library into the mutatee. This gives the user full interactive control over the mutatee.

Figure 2 shows the Dyner commands to declare a variable in the mutatee's address space, insert a conditional breakpoint, and insert code at a function's entry point.

5. Examples

There are a number of possible uses for command line instrumentation of processes. Not only does this model allow the user to analyze and respond to data produced by the mutatee, but instrumentation code, as in the case when using a regular mutator, runs with very little overhead once it has been inserted. A number of specific examples where this model may be useful follow.

5.1 Conditional Breakpoints

One of the tasks gdb [3] is very slow at is conditional breakpoints. This requires gdb to insert a regular breakpoint and catch it each time it runs to check the condition. If gdb determines that the condition is true the breakpoint is passed on to the user. With a system like Dyner, the user can merely insert a breakpoint guarded by a conditional into the mutatee and allow the mutatee to run. Once the condition is met the breakpoint will fire and control kicks back to Dyner. The user could then investigate the variables in the current function or insert or run instrumentation. The key to this is that all the code executes in the mutatee's memory space until the breakpoint actually occurs. There is not a periodic

context switch back to the mutator, making Dyner much faster than gdb for this task.

5.2 Interactive Debugging

There is a class of security applications that monitor the execution of processes for anomalous behavior. The control flow of a program is one type of behavior that is monitored [4]. Observing a program deviating from expected behavior could be a sign of a buffer overflow attack or format string attack. Either attack can have the effect of causing the program to execute malicious code injected as part of the attack. This may manifest as the program taking an unexpected control flow path during execution.

Debugging security applications meant to detect these types of attacks requires the tester to have an executable behave in anomalous ways, on demand. A full-blown mutator produced with Dyninst could attach to a target process and modify it in such a way as to trigger a security violation. However, being able to interactively cause the target program to behave anomalously can make debugging a bit easier. As the security application is interactively debugged, it can be quite helpful to have interactive control over the violations raised by the target program.

5.3 Exploratory Instrumentation

If the user is instrumenting a program for the first time, with the goal of performance measurement or program steering, the final set of mutations may not be known when the mutator is written. In the case of performance measurement to find a bottleneck, the user may need to experiment with the instrumentation, being guided by previous results when changing the instrumentation. The first set of mutations may collect data at a very high level. The entry point to each subsystem may be instrumented to determine how much time is spent in each. Once the user has data describing which subsystems account for most of the runtime, the user can remove the high level instrumentation and insert some more fine-grained instrumentation in the time consuming subsystems. The user can further hone down the cause of the bottleneck by looking at the algorithm run time and data structure access time.

The type of instrumentation needed at the lowest level may depend on the algorithms and data structures being used. The original model of running a compiled mutator and mutating a program could be quite inefficient for this exploratory task. Since a large system may have many different types of algorithms and data structures the user would need to write instrumentation for each of these before it is known which one is likely producing the bottleneck. Alternatively, the user could go through a number of cycles of building a mutator, gathering data and rebuilding the mutator. As previously noted, this may work for short running mutatees. Long running mutatees or mutatees that need a significant amount of time to warm up could present a problem.

Using an interactive instrumentation tool like Dyner allows the user to insert and remove instrumentation as the need arises. The user can insert instrumentation at a high level, let the application run and then check the gathered data. Using this data, the user can remove the previous instrumentation and insert instrumentation at a lower level as needed. This cycle can continue, until the user has discovered the source of the bottleneck.

The advantages to using Dyner are clear. The user saves the overhead of restarting the application for each cycle. In order to not restart the application using a traditional mutator, the user would have to be able to plan out all possible instrumentation code that could be needed, and exactly what conditions should trigger its use. This could be a large amount of instrumentation to write, with most of it possibly going unused. With an interactive mutator like Dyner, the user can write some instrumentation, review some data, and write some more instrumentation.

6. Related Work

The underlying mechanism used by Dyner to instrument a process, as noted above, is Dyninst [2]. Dyninst is a general-purpose runtime instrumentation library. While Dyninst is similar to binary editing tools such as ATOM [10], EEL [7] or ETCH [8] (and in fact Dyninst does have binary editing capabilities) it is more akin to tools such as Vulcan [9] and (to a lesser extent) Dynamo [1] that allow the user to instrument a running process on the fly. ATOM, EEL and ETCH all operate directly on the binary executable file offline before execution begins. Vulcan provides the user the ability to either instrument the binary executable offline or the process image while the executable is running. Dynamo acts as an interpreter for the executable file and instruments the instruction stream it is working with when necessary to perform dynamic optimizations.

The main difference between Dyner and all the systems mentioned here is that the user, while the executable is running, can create, insert and remove instrumentation code. The user can literally sit at the keyboard and write code and insert it into the process while watching the process run. None of the tools discussed here allow that level of interactivity during instrumentation.

7. Acknowledgements

Dyner has been produced by a number of people working on the Dyninst and Paradyn projects at both the University of Maryland and the University of Wisconsin. This paper is a reflection of their efforts, as well as of the authors'. We would like to specifically thank Bryan Buck for his many insightful comments during the development of this paper.

This work was supported in part by DOE Grants DE-FG02-93ER25176, DE-FG02-01ER25510, and DE-CFC02-01ER254489 and NSF award EIA-0080206.

8. References

[1] Bala, V., Duesterwald, E., Banerjia, S., Dynamo: A Transparent Dynamic Optimization System, In *Proceedings of Programming Language Design and Implementation.* June 2000.

[2] Buck, B., Hollingsworth, J.K., API for Runtime Code Patching, *Journal of Supercomputing Applications*,2000

[3] GDB, online at http://sources.redhat.com/gdb.

[4] Giffin, J.T., Jha, S., Miller, B.P., Efficient Context-Sensitive Intrusion Detection, In *Proceedings of Network and Distributed System Security Symposium*, San Diego, California, February 2004.

[5] Hollingsworth, J.K., Altinel, M., Dyner User's Guide, http://www.dyninst.org/docs/dynerGuide.v40.pdf, May 2003.

[6] Hollingsworth, J.K., Buck, B., DyninstAPI Programmer's Guide, http://www.dyninst.org/docs/dyninstProgGuide.v40.pdf , May 2003.

[7] Larus, J.R., Schnarr, E., EEL: Machine-Independent Executable Editing, In *Proceedings of Programming Languages Design and Implementation*. June 18-21, 1995.

[8] Romer, T., Voelker, G., Lee, D., Wolman, A., Wong, W., Levy, H. H., Bershad, B., Instrumentation and optimization of Win32/Intel executables using Etch, In *Proceedings of the USENIX Windows NT Workshop*, Aug 1997.

[9] Srivastava, A., Edwards, A., Vo., Vulcan: Binary tranformation in a distributed environment, Microsoft Technical Report, MSR-TR-2001-50, April 20, 2001.

[10] Srivastava, A., Eustace, A., ATOM: A systen for Building Customized Program Analysis Tools, In *Proceedings of Programming Language Design and Implementation*. May 1994.