Assignment: Final Project Report
Name: Andrew Calvano
UNI: arc2205

Deliverables

I have placed my deliverables in a GitHub repository that can be located at the following URL:

https://github.com/andrewcalvano/COMSE6156_Project

The deliverables include source code to each of the three implementations of the basic block edge coverage tool and raw data collected from atop while performing the evaluation. Also included are instructions on how to build and install each tool as well as directions on how to use them.

The README.txt file explains the repositories contents.

The INSTALL.txt file explains how to build and run each instrumentation tool.

The EVALUATION.txt file explains how to perform the evaluation.

Background Information

Dynamic Binary Instrumentation (DBI) is a technique that allows for monitoring and interacting with an application while its running. It is a different technique from debugging in that it is typically faster to use DBI based instrumentation then standard breakpoint and inspect or modify style scripts. Usually DBI is implemented in either a Just-In-Time (JIT) style compilation runtime or by inspecting or modifying CPU and memory state at execution time in an emulator. There are many DBI frameworks that are publicly available that let developers create custom instrumentation based tools. These frameworks differ in various areas such as architectures supported, granularity of instrumentation, platform support, and language of the API.

Overview

The ultimate goal of my project was to select multiple, popular DBI frameworks and evaluate them to identify the pros and cons of selecting one framework over another. Also, many of these frameworks have a steep learning curve and producing a documented tool written in each framework would be a useful starting point to another person looking to get up to speed with the framework quickly in order to produce their own tools for research or industry purposes. In addition to building a tool used in the evaluation, I measured the performance and efficiency of each DBI framework on three real world software applications. These measurements will serve as quantitative data useful for a user when deciding on which DBI framework to use. I also have attempted to identify, as well as compare and contrast, qualitative characteristics of each framework through the action of writing a functioning tool across all three frameworks.

Evaluation

Evaluation Overview

This project evaluated three DBI frameworks: Valgrind, DynamoRIO, and Intel PIN.

A sample tool was written in each of these frameworks to compare and contrast each framework. Three real world software applications: *okular*, a document viewer, *vlc*, a media player, and *tar*, an archiving utility, were used to serve as example applications to instrument. These applications vary in complexity with *okular* and *vlc* being very large, multithreaded applications using a lot of external libraries and *tar* being a more simple application that is more self contained. Data was collected from instrumenting each of these real world applications using four evaluation configurations across all three frameworks.

Evaluation Sample Tool

The example instrumentation tool was a basic block edge coverage tool that monitors an instrumented application for basic block to basic block edge transitions. At the end of the execution, all of the observed basic block edge transitions are recorded to a file for future analysis.

The tool supports instrumenting the entire address space of the application including all loaded libraries and the original executable, the constraining of the instrumented address range to a start and end address, and the ability to instrument, independently, a module or library by name.

The complexity of the tool was kept as minimal as possible so that the time constraints for this evaluation could be satisfied. The tool was also complex enough to serve as a satisfactory evaluation medium to compare each of the instrumentation frameworks.

Evaluation Configurations

Four configurations were used in the evaluation across all three DBI frameworks. For each one of the configurations and frameworks, data was collected on runtime, performance, and resource utilization. This data provided the information needed for the quantitative portion of the evaluation and allowed a comparison and contrast of the three frameworks against each other as well as against a process with no instrumentation at all.

The first configuration was the recording of data from running the three software applications independently of the DBI code. This was an attempt at collecting a baseline for which to compare the results from the executions with the DBI code in place.

The second configuration recorded each of the three software applications with the DBI framework running without any additional instrumentation code. This means that the configuration did not involve running the basic block edge coverage tool, but just the DBI engine by itself. In this configuration, the JIT engine ran the application code without inserting any additional instrumentation into the application code. This served as a baseline to compare the frameworks from a DBI runtime standpoint alone against both the data for the application’s running without DBI code as well as against the data for the configurations for the basic block coverage code.

The third configuration ran the basic block coverage tool for each framework for all code in the application including all loaded libraries as well as the main executable.

The final configuration ran the basic block coverage tool on a constrained address range corresponding to just the main executable of the evaluation application. No additional loaded libraries were instrumented in this configuration, just a smaller subset of the code.

Instrumented Applications

The *okular* application was instrumented while it loaded a test PDF file as this is typical of its normal usage since it is a document viewer application. The *okular* application was launched from the command line with the argument for the test pdf. The evaluation run concluded as soon as the GUI was loaded with the PDF file. The application would be manually closed as soon as this event occurred.

The *vlc* application was instrumented very similarly to *okular* in that a test mp4 file was used and loaded from the command line. Again, the test concluded as soon the test mp4 file was finished loading, and the application would be manually closed as soon as this occurred.

The *tar* application was instrumented during the creation of a tar.gz archive for an example directory containing four files with four bytes each in them. Each evaluation run of tar included at the end of the creation of this archive and with tar ending the test by termination.

All evaluation files (the test pdf, mp4, and archived files) are provided with the deliverables in the git repository on GitHub.

Evaluation Metric Collection

The collection of data during the evaluation runs was facilitated primarily by the *atop* utility as well as the *time* utility on Linux. The time utility was used to measure the complete runtime of the application from start to finish instrumented in the various configurations. *Atop* was used to create a log file reflecting performance and resource utilization data for each of the configurations for each application taking measurements in one second intervals.

At the end of every evaluation test the time measurement produced by the *time* utility was recorded and the *atop* log file was saved for future examination.

The evaluation runs were performed manually.

Quantitative Evaluation Results

This sections lists raw quantitative data for the evaluation performed with each framework on each application in the four configurations.

All tests were performed on a Ubuntu 14.04 x86 32-bit VM on a Macbook Air (2015). The VM had one physical CPU assigned and 4 GB of RAM.

The version of Intel PIN evaluated was version 3.2.81205. The version of Valgrind was 3.12.0. DynamoRIO was version 6.2.0. These versions reflect the most recent stable releases for each tool at the time of evaluation.

More detailed instructions on how these metrics were collected are included with the project deliverables.

Runtime Time Data

No Instrumentation

Application	**Evaluation Time**
okular	5.60s
vlc	10.801s
tar	.004s

Instrumentation Runtime Only

Application	**Intel PIN Time**	**Valgrind Time**	**DynamoRIO Time**
okular	1m 54.681s	20.000s	10.500s
vlc	2m 28.940s	32.000s	32.000s
tar	2.799s	0.569s	0.277s

Basic Block Edge Coverage (Whole Process)

Application	**Intel PIN Time**	**Valgrind Time**	**DynamoRIO Time**
okular	Timed Out (15m)	Timed Out (15m)	Timed Out (15m)
vlc	Timed Out (15m)	Timed Out (15m)	Timed Out (15m)
tar	28.500s	14.662s	10.079s

Basic Block Edge Coverage (Main Executable Addresses Only)

Application	**Intel PIN Time**	**Valgrind Time**	**DynamoRIO Time**
okular	8m 8.009s	1m 20.633s	45.033s
vlc	6m 16.409s	2m 16.456s	1m 3.586s
tar	4.558s	1.711s	0.790s

Average CPU Usage Data

No Instrumentation

Application	**Average CPU Usage (%)**
okular	24.67 %
vlc	27.91 %

tar	Failed To Capture (Finished faster than 1s)

Instrumentation Runtime Only

Application	Intel PIN Avg CPU %	Valgrind Avg CPU %	DynamoRIO Avg CPU %
okular	94.54 %	74.4 %	48.91 %
vlc	62.29 %	79.69 %	48.33 %
tar	52.5 %	44.00 %	6.00 %

Basic Block Edge Coverage (Whole Process)

Application	Intel PIN Avg CPU %	Valgrind Avg CPU %	DynamoRIO Avg CPU %
okular	3.97 %	8.21 %	2.98 %
vlc	5.42 %	50.32 %	36.38 %
tar	17.562 %	18.60 %	22.3 %

Basic Block Edge Coverage (Main Executable Addresses Only)

Application	Intel PIN Avg CPU %	Valgrind Avg CPU %	DynamoRIO Avg CPU %
okular	30.59 %	22.68 %	23.14 %
vlc	30.60 %	17.30 %	19.16 %
tar	38.20 %	23.50 %	13.0 %

Average Memory Usage Data

All percentages are based off of 4 GB of available RAM.

No Instrumentation

Application	Average Mem Usage (%)
okular	1.67
vlc	2.70
tar	Failed To Capture (Finished faster than 1s)

Instrumentation Runtime Only

Application	Intel PIN Avg Mem %	Valgrind Avg Mem %	DynamoRIO Avg Mem %
okular	2.32	3.00	2.18
vlc	3.47	4.16	3.28
tar	1.0	Failed to Capture (Finished faster than 1s)	0.00

Basic Block Edge Coverage (Whole Process)

Application	Intel PIN Avg Mem %	Valgrind Avg Mem %	DynamoRIO Avg Mem %
okular	1.66	1.62	1.00
vlc	1.70	1.08	1.77
tar	1.00	1.00	0.00

Basic Block Edge Coverage (Main Executable Addresses Only)

Application	Intel PIN Avg Mem %	Valgrind Avg Mem %	DynamoRIO Avg Mem %
okular	2.64	2.78	2.02
vlc	1.94	3.08	1.74
tar	1.00	1.00	0.00

Conclusions

Out of all of the frameworks DynamoRIO had the best results relating to time overhead, cpu utilization, and memory usage. Valgrind comes in second with timing overhead and cpu utilization only slightly lacking behind DynamoRIO. Intel PIN was by far the slowest and most resource intensive of all of the instrumentation frameworks evaluated.

There were some issues collecting measurements for tar without any instrumentation and at times even with instrumentation due to how fast the process finished. *Atop* only lets you take snapshots in 1 second intervals so if a process finishes faster than within 1s then no measurements will have been collected.

In addition, the memory utilization percentages seem slightly off and I am not sure exactly why. Regardless, the memory overhead measured does not seem to be overly large with any of the frameworks possibly making it not a great comparison metric anyway.

Qualitative Evaluation

Valgrind

Implementing the basic block edge coverage code as a Valgrind tool was the most difficult task across the three frameworks. The Valgrind code base makes heavy use of C preprocessor macros and is difficult to understand. The documentation on how to write a tool is also almost non existent, however, there are a few tools that come packaged with the source code that can be used as examples. Even with these examples I still found that I had to dive into some of the source code to figure out how to correctly structure my instrumentation tool.

In addition to this lack of documentation and non trivial learning curve, Valgrind has a fairly complicated translation process where it will take the instrumented application's code in chunks which are similar in concept to basic blocks and translate them to an intermediate language known as VEX. All instrumentation is performed on the VEX intermediate language not on the application's instructions. The combination of translated application instructions and instrumentation code are optimized and then compiled to the host instructions which may or may not be the same as the instrumented application's architecture. Because all instrumentation has to operate on the VEX instructions the learning curve is further lengthened due to having to learn how to deal with a new architectural representation.

All code for Vaglrind tools is also written in C with some preexisting code to integrate into tools but not enough to make it great from a usability standpoint. Most of the time basic C code such as linked lists, and hash tables will have to be reimplemented when writing a new tool.

My overall impression of Valgrind after writing the basic block edge coverage tool is that it supports a lot of complex things that may not be needed very often and that the framework itself is difficult to use due to the complexity some of its features, such as VEX translation, introduce. It also took me a while to actually get the code working, as I had to reference the source code of Valgrind to look up how to do certain things. The one benefit that I saw was that you could write an instrumentation tool in Valgrind that could be applied to multiple architectures. In theory, the basic block edge coverage tool I wrote was architecture independent due to it being written for VEX rather than x86 instructions. It would be an interesting experiment to see if the tool would run out of the box for a different architecture.

DynamoRIO

While DynamoRIO seems to be the most performant of the three frameworks it is also difficult to use and the documentation is lacking. Creating a tool in DynamoRIO can be cumbersome because it is built to instrument at a very low level and allows for complete rewriting of application code. While this makes for a versatile framework in what types of instrumentation tools can be created it also makes it difficult to understand exactly how to use a certain feature. It also seems like there is a steep learning curve to understand exactly how the framework should be used.

While building the basic block edge coverage tool for this framework I found myself referencing what little API documentation there was and at times source code to figure out how something was supposed to work. For example, it was not clear how to obtain the thread id for when a thread is started or stopped. I had to resort to looking at other sample tools and finally some source code to figure out how to access the event context to get the thread id I was interested in. This was painful from a usability perspective.

The framework also requires instrumentation tools to be written in C, which is a negative from a usability standpoint. This makes instrumentation code a little bit more difficult to read since it is more explicit then a more abstract language like C++.

Overall, the framework is difficult to use from a usability perspective, it lacks documentation and has an esoteric API, but it provides much lower levels of control over the application then the other two frameworks provide. It can be used to rewrite application code, or to insert arbitrary assembly instructions anywhere in the code. This is a key feature that other frameworks do not provide.

Intel PIN

Intel PIN is easily the most usable of all of the instrumentation frameworks evaluated. It is well documented and comes with many preexisting tools to reference. The API is also event based and much higher level than the other two instrumentation frameworks provide. For example, there are API calls that allow for insertion of instrumentation before and after certain events occur such as when an instruction executes, when a new image file is loaded, or when a basic block is executed.

Tools written for the framework can also be written in C++ which is a huge upgrade from a usability perspective from the other two frameworks in which tools can only be written in C. Using C++ gives tool writers access to the power of more abstract language features and libraries such as the Standard template library. While writing the basic block edge coverage tool it was actually tempting to use the STL but I wanted the evaluation to be a fair comparison across the frameworks so I stuck to the same C style implementation as the other two frameworks.

Intel PIN was the easiest framework to work with compared with Valgrind and DynamoRIO and I had very little trouble writing the tool in the framework. The API makes much more sense then the other two frameworks. In addition, it provides easy access to aspects of each disassembled instruction through a wrapper around the Intel XED disassembler library. This makes inspecting code much more intuitive than the other frameworks.

With the plethora of documentation and tools available to reference as well its awesome usability Intel PIN is my personal favorite for tool writing of the frameworks used during the evaluation.

General Conclusions

Based on the quantitative data collected as well as the higher level qualitative comparison, DynamoRIO seems to be the most performant of all of the frameworks while Intel PIN is the most usable. Intel PIN, however, had the worst performance of all of the frameworks possibly making it a good candidate to use for prototyping instrumentation tools before porting it to a more performant framework such as Valgrind or DynamoRIO. Valgrind was the least usable of all of the frameworks and was only slightly less performant than DynamoRIO. DynamoRIO's usability is not great either but because it is so performant it may make a tool writer more likely to work around its usability issues.

One of the more interesting ideas that I had coming out of this evaluation was that DynamoRIO could possibly be made a compilation target for some higher level language that makes it easy to write instrumentation tools. Since it is so low level a higher level language could make it easier to write tools targeting this framework. This could possibly overcome some of its usability and learning curve issues and make the framework more accessible to people who do not have a substantial amount of time to invest in learning the framework.

What I Learned From This Project

By doing this project I was exposed to multiple DBI frameworks and had to write tools in each of them. Through writing each of the tools I learned about different design decisions made in each of the frameworks and how these design decisions affect how tools are written in the framework. Going forward I now have a much better idea as to the pros and cons of each framework and would be more able to select a framework for a task at hand. For example, if I knew the instrumentation had to run on multiple architectures for the Linux operating system I would use Valgrind. If the tool was primarily for experimentation purposes or the tool needed to be written quickly I would use Intel PIN. If performance and overhead were the critical factor for the tool I would choose DynamoRIO. Before this evaluation I would not know where to start in selecting a framework for a particular task.

I also now have example tools written in each of the frameworks that I can reference in the future. This would save me lots of time if I need to write a new tool. In addition, because of this evaluation I am much more familiar with how each framework works and how each API is structured making me more efficient in tool writing in any of the frameworks.

References

Intel PIN
https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

Intel XED
https://software.intel.com/en-us/articles/xed-x86-encoder-decoder-software-library

DynamoRIO
http://www.dynamorio.org/

Valgrind
http://valgrind.org/

Okular
https://okular.kde.org/

VLC
http://www.videolan.org/vlc/index.html

Tar
https://www.gnu.org/software/tar/

atop
https://www.atoptool.nl/