SPEDI: Static Patch Extraction and Dynamic Insertion

Brian Fahs
CS 497 yyz Final Project Report

1 Introduction

As every modern computer user has experienced, software updates and upgrades frequently require programs and sometimes the entire operating system to be restarted. This can be a painful and annoying experience. What if this common annoyance could be avoided completely or at least significantly reduced? Imagine only rebooting your system when you wanted to shut your computer down or only closing an application when you wanted – rather than when an update occurs. The purpose of this project is to investigate the potential of performing dynamic patching of executables and create a patching tool capable of automatically generating patches and applying them to applications that are already running. This project should answer questions like: How can dynamic updating be performed? What type of analysis is required? Can this analysis be effectively automated? What can be updated in the running executable (e.g., algorithms, organization, data, etc.)? Previous works have also investigated similar topics such as the proper way to patch code into the executable, but have not focused on the actual creation of patch code and whether or not modifications are indeed “patchable”. This work will be focused more on the determination of criteria for “patchability” as well as the automated creation and application of patches to running programs. The final result of this project is a set of guidelines and pitfalls for dynamic updating and an implementation of a dynamic updating utility.

Section 2 discusses the background upon which the project was built. Section 3 discusses in more detail the idea and purpose of the project. Section 4 discusses the design and implementation of the project. Section 5 discusses the findings of the project and the experimental results. The related work to the project is covered in Section 6 and the limitations and conclusions are in Section 7.

2 Background

This project is going to be built on top of the LLVM compiler and runtime framework. The LLVM compiler framework is a complete compilation framework with a gcc 3.4 frontend and a custom backend and intermediate program representation. The executables produced by LLVM are composed of bytecodes. The bytecode format is a form of static single assignment (SSA) that can be read in by the compiler into its intermediate representation. It contains high-level constructs like symbol tables, variable types, global variables, global constants, functions, basic blocks, etc. These bytecode programs can be converted to an executable for a target machine or run inside lli, the LLVM bytecode interpreter/JIT.

3 Project Idea

This project is primarily comprised of two subsystems: dynamic program modification and program analysis. Dynamic program modification has been thoroughly researched over the last few years and there are a few different solutions. From a macroscopic level, dynamic program modification is the set of techniques required to safely modify a program while it is running. The dynamic program modifier employed in this project is a combination of some utilities already available in LLVM and some custom additions I made to the interpreter/JIT. This tool will be the medium used to apply patches to a dynamically running executable.

The other subsystem and primary focus of the project is program analysis, which is responsible for determining the changes between two different versions of a program, determining whether those changes can be safely applied to a running executable, and creating patches containing all of the necessary information for inserting the modified code. Due to the lack of previous work in this area, a major goal of the project is to produce a list of rules instructing what program modifications can be applied to running executables. The obvious other primary goal is to have a functional program extractor that is capable of automatically analyzing two versions of a program and creating dynamic program patches that can be applied through the dynamic program modifier.

The primary contribution of this work is the investigation of the potential for dynamically applying changes from later versions of executables to earlier versions without requiring the applications to restart. Through this investigation a set of guidelines will be developed to allow an automated tool to make correct decisions about which program modifications can be patched. In terms of tangible contributions, this project will produce a dynamic patching utility to automatically extract the patch code by performing a comparison between the original program and the final program. This project is not just another dynamic program modifier, but rather it is a tool set that will allow end users to create patches and apply them to running programs with little end user effort.

4 Design and Implementation

As previously stated, this project was designed on top of the LLVM compiler framework. This section is devoted to the description of those modifications.

4.1 Analysis for patching

When this project began, it was not clear whether patching would require source-level analysis, executable binary analysis, or compiler intermediate representation (IR) analysis. The first choice was deciding on what level to perform the analysis. Simple source-level analysis has the problems of not seeing the entire program at once. If files are processed one at a time, source level analysis suffers from the movement of function and definitions both within files and between files. Additionally source-level analysis suffers from ambiguities in definitions that can occur in the source language and it is difficult to create a binary patch from the source-level modifications. On the other hand, executable binary analysis makes patch creation trivial, but it can lack source-level information about types, global variables, and function names that will be required to perform the proper analysis. A compiler's IR of the entire executable bridges the gap between these two by providing an entire view of the executable that masks any aesthetic changes to the program but also provides high-level information about the executable such as types, global variables, and function names. Additionally, it is trivial to determine the patches required on the program from the compiler's IR. In this project, I chose to use the LLVM compiler framework because of its ease of use and its ability to store an executable in its IR form. The next section describes the checks that were performed on the LLVM IR for comparing two programs.

The modifications for the program analysis portion of the project involved creating a set of classes to compare and track the differences between two different programs. These classes perform several tasks where each task is used to determine not only the patch, but also whether the program can be patched dynamically. Table 1 lists the checks that are performed by these classes and Table 2 lists all of the situations where the analysis will determine that patching is not possible. In Table 2, the reason for the limitation as well as an indication as to whether this is a general limitation on patching or whether it is a limitation of the SPEDI framework.

Consistent global variable types
Consistent global variable initial values
Consistent global variable definitions (i.e., internal or external definition)
Additional global variables
Consistent function definitions (i.e., internal or external definition)
Additional function declarations
Function modifications

Table 1. Checks performed in the analysis phase of patching.

Symptom	Reason	General Limitation
Modification of the type of a global variable	If the type of the global variable is modified, then it is unclear what the result on execution would be	Yes
Modification of the initial value of a non-immutable global	Because the program will be at some unknown point in execution, it is not possible to modify the initial value of some global variable.	Yes
Addition of a non-immutable global variable	Similar to the modification of the initial value of the global, it is unclear what state that variable would be in during the middle of execution.	Yes
Movement of a variable from internal to external or vice versa	If the variable is moved to or from a shared library, it is difficult to apply this	No

	dynamically and even determine if the variable's initial value was modified.	
Addition of external function call reference	This is a limitation of this project. To perform this, the project would have to invoke the dynamic loader	No
Identical executables	Nothing required	Yes

Table 2. Cases where patching is determined not possible during the analysis phase.

4.2 Dynamic updating

The dynamic updating portion of the project was a series of additions made to lli, the LLVM interpreter/JIT. There are two requirements to perform the updates: determining whether the patch can be applied, applying the patch to the executable.

Determining whether the patch can be applied requires information about the current state of the executable because the patch cannot be applied if a function that is going to be modified is currently in the call stack. The limitation exists because it is not clear what will happen if new versions and old versions of functions interact. This is a conservative limitation imposed by the SPEDI framework. A more thorough analysis could allow modifications to still occur under certain circumstances. Determining the state of the current program requires reading the program stack for the running executable. Because lli incrementally compiles and runs the program, the symbol information for the executable does not exist, so I added functionality to look up the JIT'd function information. An additional requirement placed on dynamic updating by the SPEDI framework is that the patch cannot be applied if it was started while lli was in the process of compiling one of the source programs (This is a rare case and should not be a problem in general).

Once that it is determined that the executable can be patched, all that remains is to add/replace and recompile the functions for the executable as well as add/modify any global variables. This required some extra functionality to map all of the information (global variables, functions, etc.) from the new executable into the currently running executable. Then all modifications are inserted or copied from the new executable to the currently running version. Then all functions modified or inserted were recompiled and the entry points of the original functions were redirected to the entry points of the new functions.

In this process there a few limitations that could result in the patch not being applied. These reasons are summarized in Table 3. As with Table 2, Table 3 presents the motivation behind the limitation as well as whether the limitation is applicable to dynamic updating in general or if it is merely a limitation of the SPEDI framework. For dynamic updating a failure does not necessarily mean that the program cannot be patched at all. It means that patching should be attempted again at a different time. An improvement over this implementation would attempt to patch at multiple different points or would find a point in the program where it can be updated.

Symptom	Reason	General Limitation
Cannot patch because modified function is in call stack	Since function entry points are replaced, if the function is in the call stack and it is modified, when it returns to the function, it will return to the original. It is not clear what effect this will have on the program.	No
Cannot patch because JIT is currently JIT'ing	The JIT for lli is not re-entrant	No
Modify all or none	Only updating a portion of the changes from the new executable could result in undefined behavior	Yes

Table 3. Cases where patching is deemed not possible during the dynamic update phase.

5 Evaluation and Results

The objective of this project was two-fold: developing a dynamic updating utility and developing guidelines and evaluating the effectiveness of dynamic updating in general. To this end, we evaluate in this section (1) the functionality of the SPEDI framework and (2) the guidelines and limitations of dynamic updating.

5.1 Functionality of SPEDI

To evaluate the functionality of the SPEDI framework, I developed 16 specific test cases to test the vulnerabilities of the dynamic patching utility. Many of these cases stress multiple different features simultaneously. These test cases, their descriptions, and results are displayed in Table 4. The results in the table are the expected outcome of the test. All tests performed as expected in the framework (i.e., SPEDI works!).

Test case	Description	Result
Function_interface_1	Add and use a new argument to the function	Succeeds
Function_interface_2	Change the return value and use the result	Succeeds
Algorithm_1	Modify a for-loop in the function	Succeeds
Modify_main	Modify the function "main"	Cannot be applied
Test_identical	Update identical executable	No need to apply
Global_structure_1	Test modify global structure that is not live at the time of modification	Succeeds
Global_structure_2	Test modify global structure that is live at the time of modification	Cannot be patched
Global_variable_1	Test modify global variable's initial value	Cannot be patched

Global_structure_3	Add a global structure	Succeeds
Global_structure_4	Add to a global structure that is not live at the time of modification	Succeeds
Global_structure_5	Add to a global structure that is live at the time of modification	Cannot be patched
Insert_new_function	Add a whole new function and call it	Succeeds
Global_variable_2	Add a global variable	Cannot be patched
Global_constant_1	Add a global constant	Succeeds
Global_constant_2	Modify a global constant	Succeeds
Fix_segv	Fix a segmentation fault in a program	Succeeds

Table 4. Test cases for the SPEDI framework.

5.2 General guidelines and limitations of dynamic updating

As mentioned previously, one of the major contributions of this project is an evaluation of the limitations of dynamic patching and a set of guidelines for a dynamic patching tool. These findings are summarized in Table 5. Many of these findings have already been discussed, but the most important finding has not. The most important finding of this project is that dynamic updating of software that is currently running cannot be entirely automated in the general case. The reason for this is because high-level semantics that may only be understood by the programmer (e.g., current state stored as integers, timing of events, etc.) are difficult if not impossible for a compiler or patch extraction tool to understand without explicit help or annotations from the programmer. The fact is that if an adversary wanted to make dynamic patching succeed but cause the program to fail or perform unexpectedly, they can always succeed in doing that. As long as the high-level semantics of the program are not changed, dynamic updating should be able to succeed. Even though these limitations seem rather severe, it is my contention that many of the bug fixes and patches that are delivered over the internet are probably for minor changes such as buffer overflow detection. In most of these cases, the SPEDI type of approach will work correctly, but it will need supervision from the programmer to determine if high-level semantics have indeed changed.

Question	Solution
What type of analysis should be used?	IR-level
Can dynamic patching be automated?	Not entirely
Can algorithm updates be made?	Yes
Can organizational changes be made?	Yes
Can global data be updated?	Constants only
Can global structures be updated?	Only if they are not live at the point of patching
Can function call interfaces change?	Yes

Table 5: Guidelines and limitations in dynamic patching.

6 Related Work

There are four bodies of work that are similar in nature to this project: runtime code patching, escape analysis, online update, and code similarity analysis.

Runtime code patching [1,2,3,4] is the study of the complicated process of performing the actual insertion of code into running executables. Some of the problems that are encountered while trying to do this are (1) location – where should the instrumented code be placed, (2) free registers – what registers are free for the application to use, (3) redirection – how to redirect the program to the modified version of the code, and (4) self-deadlock – where the insertion of code into a multi-threaded program causes a deadlock within the program to occur.

In the literature, there are two different approaches to doing this: (1) interpretation and (2) using the debugging interface. The main differences are in the way that redirection is accomplished. In the interpretation approach, the running executable is run under an interpreter where it can be controlled and, therefore, redirection is relatively straightforward. The other approach uses the debugger interface to insert code into the executable and overwrite portions of the original executable to force it to enter the newly added code.

Another body of research that overlaps with this project is escape analysis [5,6]. According to [5], escape analysis is a technique to determine whether the lifetime of a variable exceeds its static scope or not. This is typically used to determine allocation policies for interpreted languages such as Objective Camel and Java. For these cases, it is beneficial in determining whether an allocation can come from the stack or whether it must come from the heap. The performance benefit of allocating data from the stack is that it will be automatically garbage collected when the function returns where as allocations from the heap must be explicitly garbage collected.

The third work that bears a significant resemblance to the work presented here is online update. Dependable systems upgrade [7] focuses on upgrading real time systems without having to bring the system down. The approach involves a new software as well as hardware architecture. All of the upgrades are performed through analytically redundant controllers. This work differs significantly from this project because the focus for this project is on upgrading existing executables without requiring any modifications to the software or hardware architecture upon which they run. Additionally, [7] focused primarily on online replacement whereas this work focuses more on determining the minimal changes between two versions of a program and dynamically applying those changes to the new system. In a somewhat similar approach, Maté [8] defines an architecture where programs can be updated on the fly for sensor devices over a wireless ad-hoc sensor network. This work too completely replaces the old program.

The last major body of work that resembles this project is code similarity analysis [9]. By the title, code similarity analysis would appear to be the solution to the program analysis proposed for this project. On the contrary, program analysis typically has the primary goal of determining if two different programs are similar enough to assume that they came from a single source. This is very useful in detecting cheating in university programming classes. However, the program analysis proposed in this work differs significantly in that it heavily leverages on the assumption that the two programs being analyzed are the same. Furthermore, the goals of the two analyses are very different.

The purpose of this project's analysis is to determine what changed and whether those changes can successfully be applied to a running executable.

7 Limitations and Conclusions

In addition to the limitations previously mentioned, there were several other minor limitations. The project does not deal with the additional complexity added from multi-threaded software, self-modifying code (SMC), or self-referential code (SRC). To handle multi-threaded software, no additional modifications should be required to the analysis phase, but the dynamic update utility would need a method to halt all threads of the application and ensure that none of the application threads are currently executing in any of the functions that are to be updated. For SMC/SRC, it is important for the dynamic update tool to be able to accurately determine the existence of such cases and avoid patching in such cases.

Another limitation of this project is in the analysis portion of the project. For simplicity, a conservative approach to patching was used. For example, any modification to the initial value of a global variable is not allowed. This is conservative, but there are cases where the programmer might want to turn on or off some functionality by dynamically updating the initial value of a variable that is only read. For these reasons, it may be beneficial to allow for programmer interaction to override certain analysis failure situations (e.g., global variable initial value changes) in cases where the programmer knows what they are doing.

In conclusion, the purpose of this project was to explore the limitations of dynamic software updating and implement a version of a dynamic update utility. Both of these have been accomplished. Through this project, all of the original questions in regards to dynamic updating have been answered. Can algorithm changes be applied? Yes. Can organizational changes be applied? Yes. Can function call interfaces (return values and parameters) be updated? Yes. Can data structures be modified? If there are provably no live variables of that data structure at the time of updating, then yes otherwise no. What type of analysis is required? IR-level. Can global data be updated? Immutable values are the only global variables that can be updated. Can dynamic patching be automated? Not really. Because of ambiguities in software programming, it is very difficult to entirely automate the process. As a result of this last finding, it will be very difficult to deploy a general tool that can guarantee that a dynamic update will be safe. However, it is possible that many of the program patches (updates, security patches, etc.) have isolated effects and do not modify the high-level semantics of the program. In these situations, dynamic updating can be a useful and valid tool.

References

[1] Z. Xu, B. P. Miller, and O. Naim. Dynamic Instrumentation of Threaded Applications, In *7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, Georgia, May 1999.

[2] A. Tamches and B. P. Miller. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels, In *Third Symposium on Operating Systems Design and Implementation* (OSDI), New Orleans, February 1999.

[3] V. Bala, E. Duesterwald, S. Banerjia. Dynamo: A Transparent Dynamic Optimization System, In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, 2000, pp. 1-12. http://citeseer.nj.nec.com/bala00dynamo.html

[4] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *1st International Symposium on Code Generation and Optimization (CGO-03)*, March 2003. http://citeseer.nj.nec.com/bruening03infrastructure.html

[5] B. Blanchet. Escape Analysis: Correctness, Proof, Implementation and Experimental Results, In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages*, pages 25-37, San Diego, CA, January 1998. http://citeseer.nj.nec.com/blanchet98escape.html

[6] A. Salcianu, M. Rinard. Pointer and Escape Analysis for Multithreaded Programs. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Snowbird, Utah, June 2001. http://citeseer.nj.nec.com/salcianu01pointer.html

[7] L. Sha. Dependable system upgrades . In *Proceedings of IEEE Real Time System Symposium*, 1998.

[8] P. Levis, D. Culler. Mate: A Tiny Virtual Machine for Sensor Networks. In *Proceedings for the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 2002.

[9] A. Aiken, http://www.cs.berkeley.edu/~aiken/moss.html