NORTHEASTERN UNIVERSITY
GRADUATE SCHOOL OF COMPUTER SCIENCE
M.S. THESIS APPROVAL FORM

THESIS TITLE: Structured Intermediate Representation Modification in the LLVM Compiler Infrastructure

AUTHOR: Leode Seth Wilson

M.S. Thesis Approved as an Elective towards the Master of Science Degree in Computer Science.

Thesis Advisor

Date 2014/04/23

Thesis Reader

Date 1/23/2014

Thesis Reader

Date 2/3 April 2014

Graduate School Approval:

Date 6 May 2014

Director, Graduate School

Original Copy Deposited in Library:

Reference Librarian

Copy Received in Graduate School Office:

Recipient's Signature

Date 5/1/2014

Distribution:
One Copy to Thesis Advisor
One Copy to Each Member of Thesis Committee
One Copy to Director of Graduate School
One Copy to Graduate School Office (with signature approval sheet, including all signatures)
Structured Intermediate Representation Modification in the LLVM Compiler Infrastructure
Thesis

by

Louis Seth Bloom

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Science

Supervised by

Assistant Professor Dr. William Robertson
College of Computer and Information Science
Northeastern University
Boston, MA
April 2014

Approved by:

[Signature]
Dr. William Robertson, Assistant Professor
Thesis Advisor, College of Computer and Information Science

[Signature]
Dr. Engin Kirda, Associate Professor
Committee Member, College of Computer and Information Science

[Signature]
Dr. Richard Carback, Principal Member of Technical Staff
Committee Member, Draper Laboratory

[Signature]
Dr. Timothy Gibson, Laboratory Technical Staff
Committee Member, Draper Laboratory

[Signature]
Mr. Neil Brock, Principal Member of Technical Staff
Committee Member, Draper Laboratory
Thesis Release Permission Form

Northeastern University
College of Computer and Information Science

Title:
Structured Intermediate Representation Modification in the LLVM Compiler Infrastructure

©2014 Louis Seth Bloom
The author hereby grants to Northeastern University and The Charles Stark Draper Laboratory, Inc. permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in any part medium now known or hereafter created.
Disclaimer

The views expressed in this article are those of the author and do not reflect
the official policy or position of the United States Air Force, Department of
Defense, or the U.S. Government.
This material is declared a work of the U.S. Government and is not subject
to copyright protection in the United States.
Dedication

To My Family
Acknowledgments

It is with immense gratitude that I acknowledge the support of Draper Laboratory. It has been an honor working with Neil Brock and Richard Carback throughout my graduate career. I also appreciate the mentorship and attention Timothy Gibson has granted me throughout my time in Boston.

Additionally, it gives me great pleasure in acknowledging the support and help of my faculty advisors, Dr. William Robertson and Dr. Engin Kirda.
Abstract

Structured Intermediate Representation Modification in the LLVM Compiler Infrastructure
Thesis

Louis Seth Bloom

Supervising Professor: Dr. William Robertson

In this thesis we propose a new approach for modifying programs expressed in an intermediate representation (IR) that leverages the many-to-one relationship between source languages and IR and also supports modifying code emitted from binary-to-IR decompilers. The open source LLVM Compiler Infrastructure offers a semantically well-defined IR and supports a variety of popular source languages and target instruction set architectures (ISAs).

LLVM currently supports program modification by translating source code into LLVM IR and performing a fixed set of transformations on the IR. This functionality is part of the LLVM optimizer and improves a program’s execution performance while preserving its visible behavior.

We improve LLVM’s techniques for systematic program modification by introducing structured methods for behavior alteration. The targets for modification are programs written in LLVM IR. First, this work presents a call graph transformation algebra which allows for call graph manipulation while preserving structural validity of a target graph. Second, this thesis explores the implementation of a structured IR editor based on this transformation algebra that allows users to modify a target program in terms of its call graph. Finally, we demonstrate the usefulness of our IR editor through an example in which we retrofit security features into the OpenSSH code-base.


## Contents

- **Disclaimer** .................................................. iii
- **Dedication** ..................................................... iv
- **Acknowledgments** ........................................... v
- **Abstract** ....................................................... vi

### 1 Introduction .................................................. 1
  - 1.1 Motivation .................................................. 2
  - 1.2 Goals ....................................................... 4

### 2 Related Work ................................................ 5
  - 2.1 IR Modification ............................................ 5
  - 2.2 Graph Abstraction ........................................ 8
    - 2.2.1 Call Graph Visualization ............................ 10
    - 2.2.2 Graph Manipulation ................................. 11
  - 2.3 Summary ................................................... 14

### 3 LLVM Background .......................................... 15
  - 3.1 Architecture .............................................. 15
    - 3.1.1 Intermediate Representation ...................... 16
    - 3.1.2 Program Structure .................................. 18
    - 3.1.3 Optimization ....................................... 19
  - 3.2 Decision to use LLVM .................................. 19
4 Methodology .................................................. 20
  4.1 Call Graph ............................................. 20
    4.1.1 Definition ....................................... 20
    4.1.2 Construction ..................................... 21
    4.1.3 Validity ......................................... 23
    4.1.4 Transformations ................................. 25
  4.2 Implementation ...................................... 32
  4.3 Summary .............................................. 33

5 Results and Evaluation ................................. 34
  5.1 Simple Transforms .................................. 34
  5.2 Case Study - OpenSSH .............................. 39
    5.2.1 OpenSSH Overview .............................. 39
    5.2.2 Evaluation Goals ............................... 39
    5.2.3 Program Point Selection ....................... 40
    5.2.4 Modification ................................... 42
    5.2.5 Evaluation and Analysis of Modification .......... 42

6 Conclusion ............................................. 45

Bibliography ............................................... 48

A Sample Reports ......................................... 52
  A.1 Sample LLVM IR .................................... 52

B Source Code ............................................. 54
  B.1 Sample Transformation Using LLVM C++ API ........ 54

C SSH Sessions ............................................. 56
  C.1 Scripted SSH Session .............................. 56
C.2  Repeated SFTP Session .............................. 57
C.3  sshd Log .............................................. 58
List of Tables

5.1  The set of functions responsible for interprocess communication in the ssh daemon  

41
List of Figures

2.1 An example graph ........................................ 9
4.1 An example call graph ................................. 21
4.2 fib.c - Call graph comparisons ...................... 26
4.3 Call graph transformation for function removal ...... 27
4.4 Call graph transformation for function cloning ...... 28
4.5 Call graph transformation for call replacement ...... 29
4.6 Call graph transformation for function replacement . 30
4.7 Call graph transformation for function wrapping .... 30
5.1 Call graph transformations on fib.c ................. 38
5.2 Privilege separation in OpenSSH .................... 41
Chapter 1

Introduction

Retrofitting security features into existing codebases can be a time-intensive task. Replacing functionality within existing code to patch vulnerabilities and monitor program behavior necessitates programmer familiarity with both program structures as well as security flaws. In addition to being costly in terms of time, this process is error-prone because there is always a possibility that a programmer may introduce some unintended behavior into a piece of software.

One approach to simplifying the task of retrofitting security features into large codebases involves modifying software in terms of a graph abstraction. For example, we can analyze a program’s code and construct from it a call graph abstraction. A call graph is a directed multigraph that represents the relationships among functions within a program. Call graphs are useful for program analysis because they can allow programmers to more easily comprehend the flow of a program. One method of call graph construction and analysis which has gained popularity over the past decade is implemented in the LLVM Compiler Framework [1]. LLVM provides mechanisms for deriving call graphs from LLVM IR modules and also offers systems for call graph manipulation.

Deriving a call graph from a target program is useful in the context of program modification because of the way we can perform algebra on graphs. With a set of algebraic transformations that permit useful manipulation of a call graph, we can effectively change the behavior of a program while abiding by the rules of the algebra. This enables us to perform deeper analysis on the target program.
We translate a program into LLVM intermediate representation (IR) before we use the LLVM compiler framework to obtain its call graph. In the typical case, we input source code into the LLVM compiler and the tool subsequently translates into IR. Programs in IR are byproducts of a compilation process more than they are standalone entities because they are in a form which abstracts the idiosyncrasies of any particular source language or platform and allows for efficient compiler optimization. These properties enable analysis of a program which is not easily replicated in high-level languages or assembly code. We use the IR to generate a call graph which we can modify in an automated fashion while abiding by the rules of an algebra.

A naive approach to IR modification forces users to manually alter individual instructions within a program. This is a poor approach because it presents an opportunity for a user to modify a program in a way such that it contains at least one invalid instruction or control flow branch; we say such a program is *structurally invalid*. We seek to ease the problem of rewriting IR by developing a more systematic approach to IR modification.

We expand other techniques for IR modification by presenting a technique which provides abstraction and safety. Our abstraction is to represent a program through its call graph. We present a call graph transformation algebra that allows users to manipulate a call graph according to a fixed set of rules. These transformations can be applied in any sequence and will always produce a program which can be compiled into a binary with equivalent behavior. This level of abstraction allows users to modify a program without having to alter individual instructions or understand low-level control flow. This model restricts users to modifying a program at the function level, and thus limits the amount of knowledge required. Our approach guarantees safety in program modification by ensuring that alterations to a program are only permitted if they will produce a program which has valid structure.

1.1 Motivation

Modifying programs at the intermediate representation level offers clear benefits over modifying programs at the source and binary levels. Perhaps
the most significant benefit offered by IR is abstraction from any particular source or machine language. This abstraction allows users to manipulate any IR program with the understanding of only one language. It is much easier for a user to develop one transformation for one language than it is to develop many equivalent transformations in multiple languages. This portability allows users to write collections of transformations once and reuse them rather than rewrite them. The transformations only need to be altered in the event that the specification for the IR changes.

Another advantage of using IR is that it permits us to easily determine a program’s control flow possibilities. This information lends itself to intraprocedural and interprocedural analyses that can be useful for systematic program transformation. For example, compilers regularly translate source code into IR in order to leverage control flow information for the purpose of program optimization. IR modification tools can utilize this information not only to provide custom optimization routines, but also to perform a fixed set of transformations that modify a program’s visible behavior.

There are a number of existing tools for structured program modification, but most of them are tied to a specific source language. Use of these tools requires access to source code as well as programmer familiarity with the source language. The matter of program modification becomes complicated when several source languages compose a codebase because modification of the codebase will require familiarity with each source language. This task could also raise the need for several tools to aid with programming in each language. Thus there is a need for a tool to translate a codebase into one common language and allow for program modification on a singular form.

One particularly useful application of an IR modification tool is the ability to retrofit some desired behavior onto an existing codebase. For example, we might want to retrofit some security features onto an application. Consider the case where an application is composed of several modules written in several source languages. Each of these modules could contain security vulnerabilities. Patching these vulnerabilities would necessitate programmer knowledge of each relevant source language as well as the know-how to fix each vulnerability. It would be more sensible to convert all the code from a given codebase into a canonical form and then perform patching on
that singular form; this could be accomplished by compiling all the source code into IR. The IR could be patched in a manner which is independent of any particular source language. Such a system could be automated to patch certain security vulnerabilities in a codebase regardless of the languages in which the codebase is written.

1.2 Goals

Our goal is to create a safe, systematic method for modifying programs written in LLVM IR. An ideal solution would provide facilities for program analysis and modification that allow users to perform a comprehensive set of manipulations to a target program. This approach is based on structured modification of a target program through manipulation of its call graph.

One of our direct sub-goals is to develop a graph transformation algebra which permits powerful program modification. Such an algebra should permit users to modify programs without knowledge of any particular source language or machine instruction set. Each transformation is kept as simple as possible in order to minimize the residual effects on the program. Keeping the effects of each operation to a minimum allows for arbitrary sequencing of transformations to create complex program adjustments.

Furthermore, our graph transformation algebra must permit operations which preserve the structural validity of the target graph. In other words, the program we are modifying through our abstraction must always remain valid and executable. The benefit of such a goal is that the user is never in a position to generate a structurally invalid non-executable program. It is important to note that our algebra will not prevent modification that can destructively modify the intended semantics of the target program.

Finally, our graph transformation algebra must power a tool which can be used to modify programs written in LLVM IR. This tool must perform validation of each transformation to ensure it is legal for the target graph. The tool must also handle all the low-level modifications to the IR program to implement the desired transformation. In other words, the tool should abstract all the changes from the user to the IR.
Chapter 2

Related Work

We draw from several existing techniques to create a new approach to program modification. This chapter provides a brief overview of related research with regard to IR modification and graph manipulation.

2.1 IR Modification

There are many existing works which utilize IR to provide users with an improved program modification experience. Some of these works depend on the availability of a program’s source code while others do not. An overview of existing IR modification tools is as follows:

Valgrind The Valgrind framework, developed by Nethercote and Seward [2], was designed to provide users with an improved interface for performing dynamic binary instrumentation. One of the ways in which it aids users is by offering a machine-independent means of altering a program’s instructions. Developed as a dynamic binary recompilation tool, it translates blocks of code from machine languages into an IR. Users perform modifications on the IR and Valgrind compiles the modified IR directly to executable code. This approach is powerful because it abstracts machine-specific idiosyncrasies and improves the tool’s extensibility for supporting more machine languages. Valgrind uses a custom IR which looks similar to machine code, so it isn’t as easily analyzable or modifiable as a compiler-level IR. In addition, Valgrind does not perform any checking to ensure that a user’s modifications result in a structurally valid program. As such, users must exercise great caution to ensure their changes effect desired behavior.
SecondWrite  O’Sullivan et al. [3] present a tool designed to retrofit security features into software for which source code is unavailable. It is similar to Valgrind in that it lets users modify programs at the instruction level through an IR abstraction. It operates by translating x86 instructions into LLVM IR and performing optimizations and security-relevant enhancements on the IR. The user can then compile the IR to executable code in any language supported by the LLVM compiler. Such an approach is useful because it ties in with existing compiler technology to perform IR modification. Thus the tool is extensible to any language which can be translated into LLVM IR. A significant downside to SecondWrite is that, like Valgrind, it lacks validity-checking schemes.

SecondWrite was designed to use the LLVM IR because it offers high-level information in an intermediate language. This design decision is an improvement over tools like Valgrind that use IR resembling assembly code because it enables users to more easily identify high-level program structures. For example, functions, arguments, and return values are all apparent in LLVM IR. Such clarity offers users greater insight into a program’s behavior and lends itself to systematic modification schemes.

Volta  Volta is a toolkit developed by Microsoft Live Labs [4] that was designed to ease the problem of developing distributed applications. Building distributed applications is challenging because programmers must implement systems for communications and data across separate program components. Volta provides automated refactoring of programs written in .NET languages in order to convert monolithic applications into distributed ones. Volta works by first translating the .NET source code for an existing program into CIL (.NET Common Intermediate Language), the .NET framework’s IR. With CIL bytecode in hand, Volta applies a series of transformations on the CIL to separate it into a functionally-equivalent, tier-separated program. The .NET framework can then be used as normal to compile the modified CIL bytecode to any supported target architecture.

By adopting an IR-level approach to automated refactoring, Volta supports modification of any program written in any number of .NET languages. Modifying code at this level also allows developers to apply other
standard transformations (i.e. optimization routines) to the CIL without interfering with Volta. Such an approach ultimately reduces the time and cost associated with distributed-application development while preserving application correctness. However, it is important to note that Volta does not offer facilities for modifying the functional semantics of a program.

**SUIF** Some compilers enable users to modify a program during the compilation process. One such compiler is the SUIF compiler developed by Wilson *et al.* [5]. SUIF consists of two components. The first component is a kernel that contains facilities for IR definitions, an API for accessing and manipulating the IR, and a system for managing compiler passes. The second component is a toolkit that implements the front and back ends of the compiler.

The SUIF IR is suited for both high-level program transformation and low-level analysis and optimization. It is in a form that can express low-level operations as well as high-level operations, including loops, conditional statements, and array accesses. In addition, SUIF’s IR manipulation API abstracts low-level changes from the user by presenting a view of IR structures in the form of C++ objects. This allows users to manipulate IR in a clean and efficient manner. Finally, the pass interface it provides enables users to modify a program at compile-time. Users can write files that specify a series of transformations that are applied to the program in an order specified by the user. Passes can easily be added, removed, and reordered, thus granting users flexibility in how they want to structure their modifications.

SUIF is not without its drawbacks. Even though its IR is flexible for representing both high-level and low-level operations, it is in the form of a massive abstract syntax tree (AST) that imposes significant overhead on the compilation process, thus inhibiting performance. Furthermore, using an AST - even a language-independent one - necessitates addition of new nodes when it comes time for the compiler to support additional front-ends. Modifying the SUIF framework to support new AST nodes is a difficult task. As such, SUIF is not an ideal solution for IR-level modification.
**DRails** Furr et al. [6] present the Ruby Intermediate Language (RIL), an intermediate representation for Ruby source code designed to ease the problem of analyzing and transforming Ruby source code. This work presents several example transformations that demonstrate the usefulness of modifying IR rather than other forms of code, including one that statically analyzes a program and removes null pointer errors. This work makes the claim that modifying a program at the IR level is thus useful for several reasons. First, RIL consolidates several source-level constructs into a singular form. Second, RIL makes the side effects of ambiguous operations much more explicit. Finally, RIL provides an explicit representation of code that would otherwise be implicit. These benefits allow for the creation of toolkits that aim to systematically alter Ruby source code.

DRails is one such toolkit that leverages the advantages of RIL [7]. DRails was designed to overcome the challenge of statically detecting and fixing type errors in Ruby on Rails, a language that supports dynamic typing. By parsing Rails source code and translating it into RIL, DRails can easily apply transformations to the program that instrument arguments passed to the Rails API. After running the instrumented program, DRails performs static analysis on the instrumentation output in order to drive transformations to make type signatures explicit. The user is then alerted to warnings regarding potentially unsafe behavior within the target program. This approach effectively implements static typing in Rails applications. Although DRails addresses a problem that is unrelated to the one we consider in our work, it still provides us with a compelling use case for systematic IR modification.

### 2.2 Graph Abstraction

Graphs are popular structures that are used to represent relationships among entities in a familiar manner. A graph can be used to express information in a way that lends itself to many common problem-solving approaches.

Figure 2.1 is a simple example of a graph. It represents borders between states in the United States. In this case, the nodes represent states and the edges represent that a border exists between two states. The states contain
labels with the abbreviated name of the state. The edges are not labeled in this example, but they could be labeled with some value relevant to the borders they represent; border length and date of establishment are sensible values for this instance.

The core of our work relies heavily on graph abstractions of programs. In our work, we use graphs to represent the calling relationships among functions in a program. Specifically, we rely on directed multigraphs in which the nodes represent functions and an edge from node $A$ to node $B$ represents a call from $A$ to $B$. Such graphs are more commonly known as call graphs.

Graphs are commonly used to represent more granular aspects of programs. For example, control flow graphs (CFGs) are directed multigraphs in which the nodes represent basic blocks, sequences of code with exactly one entry point and exactly one exit point, and the edges represent jumps among basic blocks. Such graphs are used in a variety of contexts spanning fields such as static binary analysis and optimization.

Algebraic operations can be applied to graph representations of computer
programs just as well as they can be applied to graphs representing other entities. This allows us to reason about the effects of graph manipulations upon the graph’s underlying program. Thus a graph abstraction is fitting for a program modification context.

2.2.1 Call Graph Visualization

The visual aspect of graphs is often leveraged to increase program understanding. For example, call graph visualization tools exist to provide programmers with an intuitive representation and deeper understanding of the calling relationships between functions within a program.

Stackwalker is a tool designed by Thorsten Karrer et al. to support source code navigation and comprehension through a call graph visualization [8]. Given source code as input, Stackwalker will create a visual call graph representation of the source code that users can traverse. The results presented in [8] suggest that programmers can utilize call graphs generated from source code to more efficiently locate points of interest within a program. As such, it is apparent that programmers can complete software modification tasks much more quickly with such a tool.

Although the focus of this thesis is not specifically on improving graph visualization techniques, we do offer users the option of viewing a call graph obtained through static IR analysis. We aim to provide comprehension and navigation of code at the intermediate representation level rather than the source level. The call graph we construct from an IR program consists of functions as nodes and calling relationships as edges, thus granting users a visual representation of an IR program with familiar semantics.

One of the premises of our work is that a call graph abstraction of a program written in LLVM IR will allow users to complete modification tasks more quickly and with greater comprehension than they would be able to with a linear, textual program representation. The LLVM compiler framework provides facilities for call graph construction and visualization for an IR program [9] within its optimizer component. We leverage LLVM’s call graph visualization techniques for our visualization purposes.
2.2.2 Graph Manipulation

The mathematical formality governing graphs enables us to manipulate them using algebraic transformations. Given that programs can be represented as graphs, it follows that we can manipulate programs by transforming their graphical representation. There are several useful tools that offer users a graph manipulation interface for the purpose of program modification.

**LANCET** LANCET [10] is a graphical user interface (GUI) built on top of the DIABLO binary rewriting tool [11] that enables users to apply transformations not only to a program’s individual instructions, but also to a program’s CFG. This capability enables users to more easily navigate a program to points of interest and subsequently edit the program. Examples of CFG-level modifications include adding basic blocks, splitting basic blocks, and redirecting the heads and tails of edges. Such abstract operations are not available in tools that provide only instruction-level modification because in those solutions, the instructions appear as a linear list.

The combination of instruction-level modification and CFG-level modification is a potent one because it enables users to efficiently traverse and modify target programs. However, there are three major drawbacks with LANCET’s approach. First, LANCET does not provide any guarantee of CFG validity when a user performs a transformation. This is an inherently error-prone approach to program modification. Second, LANCET forces users to modify instructions in terms of a specific machine instruction set. This limits the ease with which users can comprehend the target program. Third, modifications to the CFG are not automatically reflected at the instruction level and vice versa. This enables users to modify the CFG and the instructions asynchronously, thus making the effects of user modifications opaque. These drawbacks result in a tool which provides significant power to the user while introducing a variety of ways in which a user could generate a structurally invalid program.

**Machine SUIF** Smith and Holloway present Machine SUIF [12], a toolkit built on top of SUIF that provides an extensible framework for developing
compiler back ends. Machine SUIF includes several libraries for CFG construction and analysis that are useful for optimization purposes. The CFG construction library is interesting in a graph manipulation context because it provides an interface for modifying a program through its CFG abstraction. Specifically, the library allows users to rearrange and reconnect blocks while permitting instruction-level modification. Users are able to modify a CFG using three transformation actions: node cloning, edge redirection, and node insertion [13]. Compiler tools such as loop unrollers can make use of such actions.

Machine SUIF implements CFG manipulation in a way such that the CFG abstraction is directly linked to the underlying IR. As such, any transformations to the CFG are reflected in the program’s instructions. This is a significant improvement over tools such as Lancet that offer no such synchronization. Furthermore, Machine SUIF ensures that any transformations to the CFG will preserve the structural validity of the underlying program. This is a useful check because it ensures that the underlying program will always remain executable.

**DynInst** Buck and Hollingsworth developed DynInst [14] as a means for modifying a program’s code during execution. Rather than requiring users to edit source code, compile it, and then execute it, DynInst allows users to create snippets of code and insert them into a running program. One of the features of DynInst is that it allows users to modify the targets of existing function calls, remove function calls, and insert functions into the program. Users can invoke an inserted function by inserting a call to it. These types of modifications are essentially call graph transformations. Function removal is equivalent to node removal, function insertion is equivalent to node insertion, and modification of an existing call is equivalent to edge redirection. The authors of this work utilize these call graph transformations to demonstrate an approach for patching applications at runtime. However, they do not provide any guarantee that the transformations will preserve graph validity.
Bernat Bernat [15] developed a binary modification toolkit on top of DynInst designed to improve on tools like LANCET and Machine SUIF. First, Bernat’s work abstracts the code for a target program into an IR, thus providing the user with a platform-independent representation. This permits the user to modify programs written in many different machine instruction sets with only the knowledge of the IR. Second, the user is allowed to modify the IR in terms of the program’s CFG using a fixed set of transformations that are guaranteed to preserve the structural validity of the graph and also the underlying program. This prevents the user from making arbitrary transformations which could potentially render the underlying program invalid.

In achieving abstraction and safety, Bernat improves directly upon the features provided by LANCET and Machine SUIF. For example, Bernat’s tool only permits users to modify branches, calls, and returns through CFG modification rather than instruction modification. This is an improvement over LANCET because such restriction allows the tool to check that such modifications will preserve validity; if not, the tool will not permit the change. Bernat’s tool also improves on Machine SUIF by allowing for additional CFG transformations to include edge creation and removal. This feature is beneficial because it grants users more control in modifying the program while still providing safety.

Pin Intel’s Pin tool [16] builds upon ATOM [17] to provide an architecture-independent framework for dynamic binary instrumentation of Linux executables. Like DynInst, Pin supports function insertion and replacement through an intuitive API [18]. Users write instrumentation code in C or C++, thus limiting the complexity involved with the instrumentation. One particularly useful transform it offers is function wrapping, a form of function replacement wherein a target function $F$ is invoked by a wrapper function $F’$. $F’$ can invoke code before $F$ is invoked, after $F$ is invoked, or both. It could also invoke no additional code, but wrappers are usually not used for that purpose because such a transformation would impose unnecessary overhead. Such a transform is useful for instrumentation because it enables users to examine arguments and program state prior to a function’s invocation as well as the return value and program state after a function’s invocation.
We improve on PIN by adding function removal to our list of transforms. This enables us to remove undesired code from our program. We also allow users to perform the aforementioned transforms on IR code rather than binary objects.

2.3 Summary

There is an abundance of research into IR modification and graph manipulation. This chapter has presented two major facets of this work: the interfaces used to modify programs through an intermediate representation and the interfaces used to visualize and transform programs through a graph abstraction.

Existing tools permit program modification in terms of IR. Of these tools, the least error-prone are those that offer an integrated scheme for guaranteeing the validity of a modified program. However, some existing approaches guarantee safety by limiting changes to a program’s functional semantics. Those that do not guarantee safety allow users to alter a program’s behavior; this allows users to create an invalid program.

Current graph manipulation tools operate on the CFG and the call graph. A CFG view offers a fine level of granularity and control, but it also exposes a significant level of complexity to the user. A call graph, on the other hand, provides a much simpler representation of the relationships among the subroutines in a program. Call graph manipulation is thus fitting for cases in which a fine granularity is not required for program modification.

We address the challenges of IR modification and structural validity preservation in our approach by allowing users to modify IR through a call graph abstraction using a fixed graph transformation algebra. We draw on the approaches discussed herein to implement our own technique.
Chapter 3

LLVM Background

The LLVM compiler framework began as a research project in 2000 at the University of Illinois. Chris Lattner, then a graduate student, developed LLVM as a solution to perform lifelong program analysis and transformation [19]. It has since grown into a popular compiler used by a variety of corporate giants including Intel and Apple, Inc [20]. Its popularity can be attributed to its modularity, its open-source codebase, and its extensive list of subprojects, to name a few reasons.

In this chapter we provide an overview of the LLVM framework’s architecture. We discuss the LLVM intermediate representation, a low-level IR containing high-level information. We also delve into the optimizer component to examine the ways in which LLVM can transform programs to improve runtime performance. We then outline the different front and back ends that the LLVM framework supports. Finally, we conclude this chapter with our rationale for implementing our work in LLVM.

3.1 Architecture

The LLVM compiler framework is designed to take source code written in a number of languages as input, convert the input to LLVM IR, perform analysis and transformation on the IR, and finally generate executable code from that IR for a specific machine architecture. Several front ends exist for LLVM; each of these front ends is responsible for translating a source language to LLVM IR [21]. Likewise, many back ends exist for translating LLVM IR to machine code. The IR is the central component of LLVM because it is the focus of all program analysis and transformation.
3.1.1 Intermediate Representation

The LLVM IR is an intermediate language that is architecture independent. It is designed as the target language for aggressive optimization at compile-time, link-time, and run-time, so it contains both low-level and high-level information such as control flow transfers, memory allocations, and type structures. The IR explicitly states the type information for every value in the language and also elucidates memory allocations. It can be represented as binary, text, or an in-memory format.

3.1.1.1 Static Single Assignment

LLVM IR is written in static single assignment (SSA) form. This means that all variables in the IR are defined once. An SSA variable must always dominate all of its uses; in other words, an SSA variable cannot be used before it is assigned. SSA variables differ from source-level variables because source-level variables can be assigned to multiple times.

Consider the non-SSA code in Listing 3.1 and the equivalent SSA code in Listing 3.2. In Listing 3.1, the assignment to \( y \) depends on the value of \( x \). It is clear to a reader that the first assignment to \( x \) has no bearing on the value of \( y \) and is essentially dead code, or code that is never executed. However, a compiler must perform reachability analysis in order to eliminate such code. The code in Listing 3.2 makes it explicit that \( x_1 \) is never used, so a compiler could examine it and easily determine that Line 2 has no impact on the behavior of a program.

The LLVM compiler keeps track of SSA variable use chains in order to implement many of its transforms. Deriving such use chains is easy because any use of a variable can be traced back to a single assignment. These use chains are used to implement many of the transforms in the LLVM optimizer (i.e. dead code elimination).

```
1 x = 1;
2 x = 2;
3 y = x + 3;
```

Listing 3.1: Non-SSA variable assignment

```
1 x1 = 1
```

Listing 3.2: SSA variable assignment
3.1.1.2 Strong Typing

LLVM IR is strongly typed. Each value in the language has a type associated with it and any operation involving such a value must obey strict rules regarding the way that type can be used [22].

Consider the LLVM sub instruction. sub accepts two operands as input and returns the difference of its two operands. The operands can be either integers or vectors of integers; both operands must be of the same type. We show examples of both valid and invalid uses of the sub instruction in Listing 3.3. The load operations on Lines 5-6 explicitly state that %0 and %1 are of type i32, or a 32-bit integer. The %sub0 value is assigned the difference between %0 and %1.

The instruction on Line 8 loads a double value into %2. The sub instruction on Line 9 attempts to perform subtraction where one operand is of type i32 and the other is of type double. Such an operation is invalid according to the LLVM language specification because a double is neither an integer or vector of integers.

Type information is useful at the IR level because it presents high-level information about a program’s behavior in low-level code. This enables LLVM to easily detect when a transformation has violated the type safety of an operation as each value contains explicit information about its type. Such a strict system also permits LLVM to perform high-level analysis and transformation on low-level code. Explicit type information also improves the language’s readability as programmers can more easily track value types across multiple operations.

```
Listing 3.2: SSA variable assignment

x2 = 2
y1 = x2 + 3
```

```
3.1.1.2 Strong Typing

LLVM IR is strongly typed. Each value in the language has a type associated with it and any operation involving such a value must obey strict rules regarding the way that type can be used [22].

Consider the LLVM sub instruction. sub accepts two operands as input and returns the difference of its two operands. The operands can be either integers or vectors of integers; both operands must be of the same type. We show examples of both valid and invalid uses of the sub instruction in Listing 3.3. The load operations on Lines 5-6 explicitly state that %0 and %1 are of type i32, or a 32-bit integer. The %sub0 value is assigned the difference between %0 and %1.

The instruction on Line 8 loads a double value into %2. The sub instruction on Line 9 attempts to perform subtraction where one operand is of type i32 and the other is of type double. Such an operation is invalid according to the LLVM language specification because a double is neither an integer or vector of integers.

Type information is useful at the IR level because it presents high-level information about a program’s behavior in low-level code. This enables LLVM to easily detect when a transformation has violated the type safety of an operation as each value contains explicit information about its type. Such a strict system also permits LLVM to perform high-level analysis and transformation on low-level code. Explicit type information also improves the language’s readability as programmers can more easily track value types across multiple operations.

```
Listing 3.2: SSA variable assignment

%0 = load i32* %x
%1 = load i32* %y
%sub0 = sub nsw i32 %0, %1 // Valid operation
%2 = load double* %z
```
3.1.2 Program Structure

LLVM programs are composed of several structures, the largest of which is a module. A module contains functions, global variables, and symbol table entries. Modules can be combined into larger modules just as individual source code files can be linked together to form a whole program.

Functions are containers for basic blocks, with one of the blocks defined as the entry block. Each basic block has exactly one entry point and one exit point; thus when control is transferred to a basic block, every instruction inside the block is guaranteed to execute unless the program exits or an exception occurs. The last instruction in a basic block either returns a value or transfers control to yet another basic block.

An example LLVM module is shown in Listing 3.4. This program is a simple "Hello World" program that contains one global variable (@.str), one function definition (main), one basic block (entry), and one external function declaration (printf).

```
; ModuleID = 'hello.c'

;.str = private unnamed_addr constant [14 x i8] c"Hello World!\0A\00", align 1

; Function Attrs: nounwind uwtable
define i32 @main() #0 {
  entry:
    %call = call i32 (i8*, ...) @printf(i8* getelementptr in bounds ([14 x i8]* @.str, i32 0, i32 0))
    ret i32 0
}
declare i32 @printf(i8*, ...) #1
```

Listing 3.4: A "Hello World" program in LLVM IR
3.1.3 Optimization

The LLVM framework contains an optimizer that takes LLVM IR as input, runs a sequence of transformations on it, and outputs the optimized IR. The optimizer may also perform analysis on the IR throughout the transformation process and output those analysis results. Users can implement optimizations in LLVM using a pass system. A pass is an algorithm that can be applied to a program that transforms or analyzes it in some fashion. For example, a dead code elimination pass may perform some analysis on a program to determine what code will never be reached and then remove that code from the program. Another example is a loop unrolling pass that replaces the code for some loops with functionally equivalent independent instructions.

LLVM presents a C++ API that allows users to write optimization passes that can be inserted at various stages of the optimization pipeline. Each pass operates within a specific scope of the program (i.e. the module level, the function level, and so on) and cannot modify any part of the program that is outside of its scope. LLVM examines the properties of each pass and determines the interdependencies among them in order to efficiently schedule the passes.

3.2 Decision to use LLVM

LLVM is a modular, open-source framework. Its codebase provides us with helpful samples upon which we can structure our tools and experiments; we rely mostly on the optimizer component in order to implement our work. The optimizer draws on several high-level abstractions and demonstrates their use in order to effect powerful program transformations. Examining this code saved us time by reducing the learning curve for using the LLVM C++ API. Second, we wished to integrate into concurrent work on lifting binaries into the LLVM IR. Third, there is currently no tool for structured LLVM IR modification, so we decided to implement such a tool in order to show others how to approach this sort of problem.
Chapter 4

Methodology

In this chapter we discuss the design of our structured LLVM IR modification tool. We will examine our design decisions and provide an overview of the problems that inspired them. Our approach relies on two major components - a graph transformation algebra and a framework for implementing our transformations on existing programs. We elaborate on each of these components herein.

4.1 Call Graph

A call graph is a directed multigraph that represents the caller-callee relationships among the functions in a program. A call graph is based on two abstractions: functions and edges. Call graphs are useful for understanding function relationships because they represent high-level information about programs with no mention of low-level implementation details.

4.1.1 Definition

We define a call graph as a directed multigraph $G = (V, E)$ according to the following:

- $V$ is a finite set of nodes representing functions;
- $E \subseteq V \times V$ is a finite set of edges representing calls between functions.

We give an example call graph in Figure 4.1. The graph contains many nodes and many edges. Each directed edge $e(x, y)$ represents a call from $x$ to $y$. It is apparent from the graph that function $A$ calls $B$, $B$ calls $C$, and
so on. An edge $e(x, x)$ represents that node $x$ calls itself recursively; we see this happen at node $A$. We also see two parallel edges from $C$ to $D$; this represents that $C$ calls $D$ from two different points in $C$.

![Figure 4.1: An example call graph. The nodes represent functions (A, B, C, D) and the edges represent calls between functions.](image)

It is important to note that call graphs do not explicitly specify the order in which functions are called. For example, we see that $C$ calls $A$ once and $D$ twice. Perhaps $C$ will first call $D$, then call $A$, and finally call $D$ again; we have no way of knowing unless we actually examine the code for $C$. This opacity with respect to program flow is one of the drawbacks to representing programs in terms of call graphs. However, we choose to build our program modification tool around a call graph abstraction because we are not interested in the flow of a program so much as are we in its calling relationships. The purpose of our tool is to transform programs in terms of its function calls, so the coarse level of granularity inherent with a call graph is well-suited for our purpose.

4.1.2 Construction

Constructing a call graph from a program can be a straightforward process. Consider the C program in Listing 4.1; we will reference this program repeatedly throughout this section. Beginning with an empty graph, we can construct a call graph from this code by performing a linear sweep of the code and gathering a list of all the functions and call sites. For every function we can construct a node and for each call site we can add an edge from the containing function (the caller) to the called function (the callee). Such an approach would yield us a call graph identical to that shown in Figure 4.1.
One problem with a linear sweep approach is that it will include nodes in the graph even if they will never be called. For example, function $A$ will never invoke itself on line 4 because the conditional statement leading to its recursive call site is always false. A more precise algorithm might examine predicates to avoid inserting dead calls into the graph.

```c
void A() {
    B();
    if (1 == 2) // Always false
        A();
}

void B() {
    C();
}

void C() {
    D();
    A();
    D();
}

void D() {
    void (*foo)();
    foo = (&A);
    (*foo); // Equivalent to A();
}
```

Listing 4.1: Example C functions

Another issue that poses a problem for call graph constructors is pointer aliasing. Consider line 20; function pointer $foo$ has been assigned the address of $A$, so $A$ is invoked when the code on line 20 is executed. This type of function invocation will only be detected by a more advanced algorithm. Now suppose that $foo$ is assigned to a function pointer which is passed as a parameter to $D$ from a function in a different program. In that case, there would be no way to statically determine what function or set of functions $foo$ may call; the callee could theoretically be any function in the program [23].

The problems presented by predicates and function pointers may cause some call graph construction algorithms to generate a graph that is conservative, incomplete, or both. We address these issues during our call graph construction phase by augmenting a linear sweep algorithm with various
static analysis tools included within the LLVM optimizer framework [9]. The LLVM optimizer includes functionality that we can leverage in order to eliminate dead code and resolve function pointers. The call graph we construct is thus more accurate than it would be if we utilized only a linear sweep algorithm.

### 4.1.3 Validity

Program modification is complicated by the risk that a user will alter a program in such a way that the program contains an illegal sequence of instructions. Structured IR modification addresses this problem by providing an interface for a user to modify a program while ensuring that the user can never generate an invalid program. Our approach abstracts a program in terms of a call graph so a user can transform the program in terms of the graph and then generate an executable version of the underlying program.

We say that a call graph is valid if it represents a valid program. The strength of this approach is that we create a set of rules that define a valid call graph; any transformation we apply to our graph must preserve graph validity or we do not allow it. The program backing the graph is thus always guaranteed to remain valid as well. It is important to note that structural validity indicates that a program contains only legal instruction sequences; there is no implication about preservation of runtime behavior.

#### 4.1.3.1 Structural Validity

We draw from LLVM’s definition of well-formedness in order to formulate a definition for structural validity of programs written in LLVM IR. The LLVM language reference [22] elaborates on the criteria for well-formedness for every structure in the LLVM IR. The list is rather lengthy, but we are primarily concerned with the rules for well-formedness with respect to functions. For example, LLVM IR functions cannot accept a parameter with a void type. They also cannot have call argument types that differ from their prototype.

We define an LLVM IR program $P$ as a set of instructions $I$ where $I = (i_0, i_1, ..., i_n)$. We say that $P$ is structurally valid if and only if there does
not exist an instruction in $I$ that violates the well-formedness of $P$.

Consider the instruction in Listing 4.2. This instruction is not well formed because it violates the SSA property of the LLVM IR; $\%n$ cannot be assigned to the result of a subtraction operation dependent on the value of $\%n$. Any program containing this instruction would not fulfill our criteria for structural validity. Our work assumes that any input program is structurally valid; we do not consider the case where an input is structurally invalid.

<table>
<thead>
<tr>
<th>%n = sub i32 10, %n</th>
</tr>
</thead>
</table>

Listing 4.2: Structurally invalid LLVM IR

### 4.1.3.2 Call Graph Validity

A call graph $G$ is structurally valid if and only if every node in $V$ represents a well-formed function and every edge in $E$ represents a well-formed call. We assume we always receive structurally valid programs as input, so it follows that any call graph we construct from a given input program will also be structurally valid.

There are many ways in which we could modify a structurally valid call graph such that the underlying program would become structurally invalid. For example, let us consider the program in Listing 4.3. This program contains a function $\text{fib}$ that takes an integer $n$ and returns an integer representing the $n$th number in the Fibonacci series. When the program executes, it outputs $\text{fib}(15)$. The call graph for this program is shown in Figure 4.2(a).

Consider a scheme in which users can modify a call graph without restriction. Such a scheme would allow the user great control over the function-level behavior of the underlying program. However, it is not hard to imagine a transformation to the graph in Figure 4.2(a) that would render the underlying program structurally invalid. For example, a user could redirect an edge $e(\text{fib}, \text{fib})$ to point to $\text{invalidFib}$, thus creating an edge $e(\text{fib}, \text{invalidFib})$ as shown in 4.2(b). Such a call would introduce structural invalidity into the graph because the result of the call is used in an addition operation. $\text{invalidFib}$ returns void, so it is an invalid operand to the binary expression on Line 14 in Listing 4.3. If the graph is structurally invalid, then we know
that the underlying program is also structurally invalid.

Instances such as the one described above are our primary motivation for developing an approach with inherent safety. By limiting the user’s transformations to a fixed set of operations that preserve structural validity, it becomes impossible for the user to create an invalid program. However, this limits functionality by forcing users to modify a program within the scope of the available operations.

```c
#include <stdio.h>

void invalidFib(int n) {
  // Do nothing
}

int fib(int n) {
  if (n == 0)
    return 0;
  if (n == 1)
    return 1;
  return fib(n - 1) + fib(n - 2);
}

int main() {
  int x = fib(15);
  printf("%d\n", x);
  return 0;
}
```

Listing 4.3: fib.c - A Recursive Fibonacci Series Program in C

### 4.1.4 Transformations

The primary advantage of call graph abstraction in our work is that it enables us to develop an algebra for modifying the call structures in LLVM IR programs. Our call graph transformation algebra consists of a set of graph transformations $T$ and a call graph structural validity constraint $C$. Our algebra contains four types of transformations: removal, cloning, replacement, and insertion. A transformation $t$ modifies a target graph $G$ by replacing a subgraph of $G$, $A$, with a subgraph $B$. We consider a transformation to be valid so long as a structurally valid input graph is not rendered
structurally invalid as a result of the transformation. Users can arbitrarily sequence these transformations on a structurally valid input graph and will always produce a structurally valid output graph. We define each of this transformations in this section.

4.1.4.1 Removal

Function removal removes a function $f$ from the graph by eliminating $f$ as well as all out-edges from $f$. We do not allow the removal of $f$ if $f$ has any interprocedural in-edges because such a transformation would leave dangling edges in the graph. A dangling edge is indicative of a call to a non-existent function; such a call would render the underlying program structurally invalid. We allow removal of intraprocedural in-edges because they only refer to the function we are removing. We present an algorithm for function removal in Algorithm 1. We show an example of a function removal transformation in Figure 4.3.
Algorithm 1 An algorithm for removing a function $f$ from a call graph. $f$ cannot have any interprocedural in-edges. We remove $f$ by removing all of its out-edges from the graph. We then remove the node for the function.

1: procedure REMOVE($G, f$)  \Comment{The call graph and the function to remove}
2: \hspace*{1em} for each out-edge $e$ from $f$ do
3: \hspace*{2em} EdgeRemove($G, e$)
4: \hspace*{1em} end for
5: \hspace*{1em} NodeRemove($G, f$)
6: end procedure

![Call graph transformation for function removal](image)

Figure 4.3: Call graph transformation for function removal. In this case, we remove $f$ from the call graph. We also remove all of $f$’s out-edges from the graph.

4.1.4.2 Cloning

Function cloning adds a function $f'$ to the graph where $f'$ is a clone of some function $f$. $f'$ will be initialized with $f$’s set of interprocedural out-edges. Each edge $e(f', f)$ will then be redirected to become $e(f', f')$. $f'$ will initially have no interprocedural in-edges. We present an algorithm for function cloning in Algorithm 2. We show an example of a function cloning transformation in Figure 4.4.
Algorithm 2 An algorithm for creating a clone $f'$ of a function $f$ in a call graph. $f'$ will be identical to $f$ except all edges $e(f', f)$ will be replaced with $e(f', f')$.

1: procedure CLONE($G, f$) \Comment{The call graph and the function to clone}
2: \hspace{1em} NodeClone($G, f$) \Comment{Create a node $f'$}
3: \hspace{1em} for each edge $e(f, b)$ do
4: \hspace{2em} if $b \neq f$ then
5: \hspace{3em} EdgeInsert($G, e(f', b)$) \Comment{All interprocedural calls remain the same}
6: \hspace{2em} else
7: \hspace{3em} EdgeInsert($G, e(f', f')$) \Comment{Intraprocedural calls are redirected to $f'$}
8: \hspace{2em} end if
9: \hspace{1em} end for
10: end procedure

Figure 4.4: Call graph transformation for function cloning. In this case, we create a clone $f'$ of function $f$. Note that $f'$ has no in-edges.
4.1.4.3 Replacement

Call Replacement  Function call replacement redirects an edge in the call graph from \( f \) to \( f' \). We only allow call replacement when \( f \) and \( f' \) have identical signatures. We show an example of call replacement in Figure 4.5.

Function Replacement  Function replacement redirects all edges in the call graph from \( f \) to \( f' \). We only allow function replacement when \( f \) and \( f' \) have identical signatures. We show an example of call replacement in Figure 4.5.

Function Wrapping  Function wrapping replaces calls to \( f \) with calls to \( f' \) where \( f' \) is a wrapper function containing calls to \( pre \) (optional), \( f \), and \( post \) (optional), in that order. \( f' \) must have the same signature as \( f \) because it accepts the same parameters as \( f \) and returns a value of the same type as \( f \). We pass \( pre \) the same arguments that we pass to \( f \), so we require that it accepts the same parameter types as \( f \), and in the same order. If \( f \) has a void return type, then we call \( post \) with no arguments. Otherwise, we pass \( post \) the return value of \( f \), so we mandate that it must accept only one argument, and that argument must be of the same type as the return value. After the optional call to \( post \), \( f' \) returns the return value of \( f \).

Function wrapping enables us to effectively instrument function arguments and return values. We present an algorithm for function wrapping in Algorithm 3. We show an example of function wrapping in Figure 4.7.

![Figure 4.5: Call graph transformation for call replacement. In this case, we replace a call to \( f \) with a call to \( f' \).](image)
Figure 4.6: Call graph transformation for function replacement. In this case, we replace a function $f$ with $f'$.

Figure 4.7: Call graph transformation for function wrapping. In this case, we wrap a function $f$ with a pre-invocation call to $pre$ and a post-invocation call to $post$. 
Algorithm 3 An algorithm for wrapping a function $f$ within a call graph. Every invocation of $f$ is replaced with an invocation of a wrapper function $f'$.

1. **procedure** \textsc{Wrap}$(G, f, \text{pre}, \text{post})$  \Comment{The call graph, the wrappee, the pre-invocation function, and the post-invocation function}
2. NodeInsert$(G, f')$  \Comment{Create a node $f'$}
3. $f' = \text{GenerateWrapper}(f, f', \text{pre}, \text{post})$
4. Replace$(f, f')$  \Comment{Replace all calls to $f$ with calls to $f'$}
5. **end procedure**

6. **procedure** \textsc{GenerateWrapper}$(f, f', \text{pre}, \text{post})$
7. $\text{Args} = f$’s argument list
8. $\text{RetType} = f$’s return type
9. $f''$’s argument list = $\text{Args}$
10. $f''$’s return type = $\text{RetType}$
11. \textbf{if} $\text{pre} \neq \text{null}$ \textbf{then}
12. \hspace{1em} AddInstruction$(f', \text{call pre with Args as parameters})$  \Comment{Pre-invocation call}
13. \textbf{end if}
14. AddInstruction$(f', \text{call } f \text{ with Args as parameters})$  \Comment{Original call}
15. $\text{retValue} = f$’s return value
16. \textbf{if} $\text{post} \neq \text{null}$ \textbf{then}
17. \hspace{1em} AddInstruction$(f', \text{call post with retValue as the parameter})$  \Comment{Post-invocation call}
18. \textbf{end if}
19. AddInstruction$(f', \text{return retValue})$  \Comment{Return wrappee’s return value}
20. **end procedure**

4.1.4.4 Insertion

Function insertion creates a function $f$ in the graph. It initially has no in-edges. We construct a set of out-edges from $f$ based on its calls. We preserve structural validity by requiring all calls to be structurally valid. For example, $f$ cannot have any calls to functions that are not already present in the call graph. We show an algorithm for function insertion in Algorithm 4.
Algorithm 4 An algorithm for inserting a function \( f \) into a call graph. We iterate over each call in \( f \) and add an edge to the graph to represent that call.

\[
1: \textbf{procedure} \ \text{INSERT}(G, f) \quad \triangleright \text{The call graph and the function to insert} \\
2: \quad \text{NodeInsert}(G, f) \quad \triangleright \text{Create a node } f \\
3: \quad \textbf{for each} \ \text{call } c \ \text{in } f \ \textbf{do} \\
4: \quad \quad \text{EdgeInsert}(G, e(f, \text{callee})) \\
5: \quad \textbf{end for} \\
6: \textbf{end procedure}
\]

4.2 Implementation

We implemented our structured call graph editing techniques in the LLVM 3.4 compiler infrastructure by building upon the optimizer component of the framework. The optimizer is built on top of an open-source C++ API and is designed to modify LLVM IR programs while ensuring behavior preservation. The C++ API contains abstractions for every structure in the LLVM IR. We examined the optimizer code and learned how we could leverage the C++ API abstractions to implement a program modification tool.

We expanded upon LLVM’s program modification components by allowing users to modify IR programs according to the rules of our graph transformation algebra. The primary challenge in doing this was determining what changes we had to make to the IR to reflect changes to the call graph. The C++ API grants us the capability to modify the IR code through abstract structures, so we utilized these structures to perform the IR-level changes for us. As such, we never wrote a single line of IR code; all of our code is written in C++. A sample of the source code required to implement our transforms is presented in Appendix B.1.

One of the challenges we faced was determining how users can write IR code that is to be injected into the program during function wrapping and insertion. We decided it would be best to simply leverage the existing LLVM front ends to do the source-to-IR translation. For example, if a user wants to insert a function into a program, then he can write it in C or C++ and then translate it to LLVM IR using clang [24]. The resulting IR can then be linked into the target program.
4.3 Summary

In this chapter we presented an approach to LLVM IR modification based on call graph manipulation. We defined structural validity and outlined ways in which we can modify call graphs while preserving that validity. Using the four transformations we described - removal, cloning, replacement, and insertion - we can modify the behavior of existing programs. In the next chapter we demonstrate the usefulness of our tool through a real-world example.
Chapter 5

Results and Evaluation

In this chapter we walk through example use cases of our tool. We begin by demonstrating simple program modification examples and conclude with an application of our tool in which we retrofit additional security features into OpenSSH.

5.1 Simple Transforms

In this section we demonstrate some simple examples of our tool’s capabilities. We perform all of our example transformations on the program shown in Listing 5.1, fib.c. This program simply computes \( f_{ib}(45) \), the 45th Fibonacci number, and prints the result to the standard output. See Appendix A.1 for the equivalent IR for this program.

```
#include <stdio.h>

int fib(int n) {
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
    return fib(n - 1) + fib(n - 2);
}

int main() {
    int x = fib(45);
    printf("%d\n", x);
    return 0;
}
```

Listing 5.1: fib.c - A Recursive Fibonacci Series Program in C
```c
int dynamicFib(int n)
{
    if (n < 2)
        return n;

    int f[n+1];
    f[0] = 0;
    f[1] = 1;
    for (int i = 2; i <= n; i++)
        f[i] = f[i - 1] + f[i - 2];
    return f[n];
}
```

Listing 5.2: dynamicFib - A function for computing the nth Fibonacci number using dynamic programming.

We begin by converting our program into LLVM IR. This is trivial as we can simply use clang [24] to perform the translation for us. With the IR in hand, we can use our tool to alter it. The call graph for this program is shown in Figure 5.1(a).

Our next step is determining a point in our program at which we will perform modification. Looking at the `fib` function, we determine that we can replace it with an equivalent function. Such a replacement function is shown in Listing 5.2. In order to insert this function into our program, we convert it to IR using clang. We then use our tool to insert it into the module; we see the effect of this transformation in Figure 5.1(b). Next, we use our function replacement transformation to redirect all edges to `fib` to `dynamicFib`, as shown in Figure 5.1(c). Moving on, we see that there are no remaining calls to `fib`. We can remove this function from the call graph and yield the resulting graph shown in Figure 5.1(d). At this point in our program transformation, we have simply refactored fib.c; we have not altered its visible behavior in any way.

Let us consider a case where we would like to wrap a function. We could do this in fib.c in order to log the arguments passed to `dynamicFib` and also to log `dynamicFib`'s return value. Let us consider the functions `pre` and `post` shown in Figure 5.4. We can use our wrapping transformation to place `dynamicFib` inside of another function `dynamicFibWrapper` that
prefaces every invocation of `dynamicFib` with a call to `pre`; this call will be made with the same value of `n` that is passed to `dynamicFib`. `dynamicFibWrapper` will postface every call to `dynamicFib` with a call to `post`; this call will be made with the return value of `dynamicFib`. The call graph displaying the result of this wrap is presented in Figure 5.1(e). The LLVM IR for this wrap is shown in Listing 5.3.

```llvm
define i32 @dynamicFibWrapper(i32 %n) {
  entry:
    ; Calls pre using the same arguments that are passed to the
    ; wrapped function
    call void @pre(i32 %n)
    ; Calls the wrapped function and assigns the return value to %0
    %0 = call i32 @dynamicFib(i32 %n)
    ; Calls post using the wrapped function's return value
    call void @post(i32 %0)
    ; Returns the wrapped function's return value
    ret i32 %0
}
```

Listing 5.3: LLVM IR for a function wrapper.

At last we have a program that has undergone several transformations. We have successfully refactored fib.c to utilize a more optimized function for computing the `n`th Fibonacci number. We have also implemented a primitive logging system for outputting `n` as well as the `n`th Fibonacci number. This simple example demonstrates that our structured LLVM IR modification tool permits easy program modification. We could have written the functions we injected in any source language so long as we had a front end to convert it to LLVM IR. It is important to note that we never wrote a single line of IR; we just leveraged existing translators to obtain the IR and then relied on our graph transformations to do all of the program modification for us.

Although this example is admittedly trivial, it emphasizes the abstraction and safety provided by our approach. We demonstrate a more compelling use case of our tool in the next section.

```c
void pre(int n) {
  printf("%d was passed to dynamicFib!\n", n);
}
```
Listing 5.4: \textit{pre} and \textit{post} will be invoked immediately before and immediately after \textit{dynamicFib} is invoked, respectively.
(a) Call graph for fib.c

(b) First, we insert dynamicFib into the call graph.

(c) Next, we replace fib with dynamicFib.

(d) We remove fib from the graph as it is no longer referenced.

(e) We wrap dynamicFib with calls to pre and post. Note that dynamicFib’s recursive calls have been routed through dynamicFibWrapper as a result of the wrap transformation.

Figure 5.1: Call graph transformations on fib.c. The figures show the effects of our call graph transformations on a simple program.
5.2 Case Study - OpenSSH

In this section we provide an overview of a study we performed in order to evaluate the usefulness of our approach to program modification. We describe OpenSSH, our target codebase written entirely in C, as well as our course of action for implementing changes to its behavior.

5.2.1 OpenSSH Overview

OpenSSH [25] is a software library that provides network communication security via encryption using the SSH protocol. It is comprised of several popular programs including, but not limited to, sftp, scp, sshd, and ssh. This suite provides users with secure replacements for programs like telnet and ftp by providing confidentiality and authentication. All login credentials are passed across the network in encrypted form rather than plaintext. Additionally, host keys are used to ensure that clients have a way of verifying the identity of the system with which they are communicating.

We targeted OpenSSH for modification because it is a relatively large codebase, it is open-source, and because it is relevant to the security domain. We were concerned with operating on a sizable codebase in order to demonstrate the utility of our work in modifying nontrivial programs. We opted for open-source software because it enabled us to easily generate LLVM IR from it. Finally, we wanted to work with software that is security-relevant because it provides us with a compelling example for the application of our work in a real-world scenario.

5.2.2 Evaluation Goals

Our approach for our OpenSSH modification is threefold. First, we aim to retrofit security features into the OpenSSH codebase. When we mention OpenSSH, we refer specifically to OpenSSH version 6.4.1. Our second goal is to apply our security features to OpenSSH by identifying points of interest at the function level and modifying them using our toolkit. Our third and final goal is to establish a system for implementing patches written in LLVM IR.
The first step toward achieving these goals was obtaining a copy of OpenSSH programs in LLVM IR form. We accomplished this by using clang to generate an equivalent LLVM IR module along with each executable during the compilation process. This allowed us to examine an entire program such as ssh in the form of one LLVM IR module. We then pinpointed functions of interest and modified the program to instrument those functions. This essentially allowed us to set up a logging system. Finally, we created a system for implementing patches written in LLVM IR by inserting references to external functions which users can resolve with their own implementations of those functions.

5.2.3 Program Point Selection

We began our program modification process by identifying points in our program that were of interest to us. As we were looking for ways to improve the security features of OpenSSH, we delved into the areas of the codebase that were relevant in a security context.

One interesting aspect of the OpenSSH codebase is that some portions of it are written using privilege separation, a security technique that separates a single program into two separate programs: a slave that drops its privileges and a monitor that fulfills requests on behalf of the slave [26]. The monitor is forked from the slave process at the time the program executes; the two processes then communicate via pipes. If an attacker overruns the slave and attempts to execute some set of commands with elevated privileges, the attacker will be limited by the monitor as it is the final authority on what operations will actually execute. We present a visual representation of the privilege separation model in OpenSSH in Figure 5.2.

The privilege separation architecture in OpenSSH consists of roughly 1500 lines of code and is designed to protect the ssh daemon from attack [27] [28]. As there is a clear security concern regarding the messages being passed between the slave and monitor processes in the ssh daemon, we deemed the privilege separation boundary connecting these two process to be a compelling target for our case study.
Our program point selection led us to identify the functions responsible for passing messages between the monitor and slave processes. A cursory examination of the source code for the ssh daemon revealed that three functions are ultimately responsible for interprocess communication (IPC): \texttt{mm\_request\_send}, \texttt{mm\_request\_receive}, and \texttt{mm\_request\_receive\_expect}. We show the number of caller functions that invoke each of these functions in Table 5.1. These three functions do not perform context-sensitive operations, but their callers do. As such, we are interested in all of the functions that call the three functions responsible for IPC.
5.2.4 Modification

With our program points of interest selected, we developed a plan for modifying the ssh daemon in some useful way. Specifically, we opted to demonstrate the utility of our toolkit by retrofitting a modular instrumentation framework into the ssh daemon; such a framework is helpful for tasks such as logging and performance measurement.

We implemented our instrumentation framework by running an automated script that wrapped our target functions with calls to unique external pre-invocation and post-invocation functions. In other words, for each function \( x \) in our list of functions of interest, we wrapped \( x \) with calls to an externally-defined function \( pre_x \) and an externally-defined function \( post_x \) according to the rules of our function wrapping transformation. Our algorithm for this process is shown in Algorithm 5. We show a sample of the LLVM IR generated by our algorithm in Listing 5.5.

```llvm
1 declare void @pre_mm_answer_rsa_response(i32, %struct.Buffer*)
2 declare void @post_mm_answer_rsa_response(i32)
3 ; Function Attrs: nounwind ssreq uwtable
4 define i32 @mm_answer_rsa_response-wraper(i32 %sock, %struct.Buffer* %m)
  #0 {
5 entry:
6 call void @pre_mm_answer_rsa_response(i32 %sock, %struct.Buffer* %m)
7 %0 = call i32 @mm_answer_rsa_response(i32 %sock, %struct.Buffer* %m)
8 call void @post_mm_answer_rsa_response(i32 %0)
9 ret i32 %0
10 }
```

Listing 5.5: Sample LLVM IR generated by an automated algorithm for modular framework establishment.

5.2.5 Evaluation and Analysis of Modification

The usefulness of our tool became overwhelmingly apparent during our testing. Our automated wrapping of all functions that invoke the IPC functions in the ssh daemon executed in an average of 0.63 seconds across 10 executions. Performing the same operations by hand is significantly more intensive in terms of time and precision. In order to manually implement
Algorithm 5 An algorithm for retrofitting a modular instrumentation framework into OpenSSH.

1: procedure INSTRUMENT(G, F) $\triangleright$ The call graph and a list of functions to instrument
2:  Callers = *Empty*
3:  for each function $f$ in $F$ do $\triangleright$ Gather a list of distinct callers to the functions in $F$
4:    if !Callers.contains($f$) then
5:      Callers.add($f$)
6:  end if
7: end for
8:  for each function $f$ in Callers do
9:    ExternalNodeInsert($G, \text{pre}_f$) $\triangleright$ Create a reference to an externally-defined
10:   node for a function to be invoked before each call to $f$
11:   ExternalNodeInsert($G, \text{post}_f$) $\triangleright$ Create a reference to an externally-defined
12:   node for a function to be invoked after each call to $f$
13: end for
14: end procedure

the same program transformations, we would have to manually create each wrapper function and replace all uses of each wrappee function with the corresponding wrapper. This task is time-intensive because there are many functions spread throughout many source files that invoke the wrappees. We have to search source files to find all of the callers and update the call sites to our wrappee by calling the wrapper instead. Manual call replacement is made complicated by function invocation via pointers. We were able to implement function wrapping in approximately 90 seconds for some functions, but others took upwards of 360 seconds. We conservatively estimate that it would take four hours to perform the same wrapping by hand. It is clear that an automated process for implementing function wrapping is preferable to manual wrapping because it is both fast and correct.

We manually examined the modified ssh daemon in order to ensure the automated transformation to the IR behaved as expected. Specifically, we checked that the resulting executable’s call graph respects the rules that we defined in our methodology. We then used llvm-link [29] to link definitions that we created for a subset of the externally-referenced functions. Next, we again examined the ssh daemon to ensure that our function definitions linked properly to the modified ssh daemon. If we were to implement definitions
for each of the pre/post-invocation functions, we would be able to generate an executable version of the modified ssh daemon.

We performed a dynamic analysis of our transformation in order to show that the executable version of a modified sshd program would be affected by our transformations by using our method to wrap many of the functions in the ssh daemon. We also wanted to show that the functional behavior of programs interacting with a modified ssh daemon would not change. We performed three automated experiments over the original and modified sshd programs in order to make this point. First, we ran a single scripted SSH session as shown in Listing C.1; it generated the output we show in Listing C.2 for both the original and modified program. Second, we ran the same SSH session ten times in a row using the same script; it generated the same expected output all ten times for both the original and modified program. Finally, we ran a scripted SFTP session as shown in Listing C.3 ten times in a row; it performed correctly all ten times for both the original and modified program. We show the output for this scripted SFTP session in Listing C.4.

Our automated sessions show that the scripted sessions behaved identically and as expected every time for both the original and modified program. However, we wanted to demonstrate that our wrapping transformations had actually modified the SSH daemon in some way. As such, we simply implemented our wrapper functions to print output to sshd’s integrated debugging framework. In Listing C.5 we show the output from a modified sshd session using the script from Listing C.1. We can see several instances in which our wrapper functions were called; for example, Lines 37 and 40 show that wrapper functions were called and also that arguments and return values were captured. This output demonstrates that our transformations to sshd took effect and also that programs interacting with sshd behaved properly.
Chapter 6

Conclusion

We have shown that it is possible to automatically retrofit security features into an existing codebase by applying call graph transformations to LLVM IR programs. Other program modification techniques may provide finer granularity for modification, but the use cases for such tools are different from the ones discussed herein. Other solutions geared toward modifying IR programs typically only offer refactoring capabilities, and those that offer behavior modification rarely offer structurally safe and abstract facilities for transforming a target program. We demonstrated a technique that is both abstract and structurally safe.

Our demonstration of automated transformation to the OpenSSH library does not imply that our method is limited just to OpenSSH programs. Our technique is applicable to any LLVM IR program, regardless of the source language or binary used to derive the IR. All we need to use our toolkit is an LLVM IR module. In other words, if we can decompile an executable to IR or compile source code to IR, we can modify it using our method.

Our work is built entirely upon the LLVM library components responsible for program optimization. Although LLVM offered facilities for modifying programs at the CFG level and the instruction level, we did not take advantage of them for our work because we were only concerned with call graph modification. Implementing CFG modification capabilities would improve the program modification techniques discussed in this work. For example, if instrumentation could be implemented at the CFG level, then users could instrument specific call sites based on knowledge of branches taken or not taken. Another advantage of CFG modification would be the capability to redirect the targets of jumps between basic blocks. Such modification
would enable users to control program behavior inside of functions. An interface for performing CFG modification could be established on top of our existing code base and improve the facilities users have for LLVM IR modification.

It is relatively easy to grasp the relationships among functions in a call graph with few nodes. However, large graphs can be difficult to grasp through a cursory examination. Although our tool allows users to apply transformations to a call graph abstraction of a program, the user is ultimately responsible for identifying the targets of modification. We find our tool most useful when it is applied as an automated executor of some high-level specification such as ”Instrument the parameters and return value of all functions that call some given function.” Modifying a codebase such as OpenSSH according to such a specification is time-intensive and error-prone, but our toolkit can perform all the necessary operations for such a modification both quickly and correctly.

Our case study demonstrates that we implemented a modular instrumentation framework capable of capturing the parameters and return values of a large set of functions. As we showed in the case study dealing with OpenSSH, our toolkit does not define the instrumentation functions; the toolkit just inserts calls to them. It is the user’s responsibility to implement the instrumentation functions. Such a system is useful for security purposes because it enables users to implement instrumentation separately from the codebase undergoing instrumentation. For example, a codebase may be distributed with the calls to external instrumentation functions present, but the instrumentation functions may not be defined. A separate party may then be responsible for implementing the instrumentation in a linking step. The instrumentation functions could then be used for logging security-sensitive portions of a program.

This thesis work improves upon the current state-of-the-art with regard to IR program modification. However, there are other components that could extend the capabilities of this work.

Adding control flow graph modification capabilities would allow users a finer level of granularity for program modification. Although the requisite knowledge of program control flow is greater for CFG modification, such
a system would enable users to make more fine-tuned changes to a target. A toolkit that allows users to systematically modify IR at the CFG level would permit behavior modification within a function without the need for function replacement.

Our work could also be extended to permit modification of a program during execution. We currently only permit static modification of a program. Dynamic program modification would allow users to modify program behavior at runtime; such a capability would be useful for ”hot patching” a program. In such a system, the user should be able to transform the code for a program and have the transformations take effect on the executable instantly.
Bibliography


Appendix A

Sample Reports

A.1 Sample LLVM IR

```llvm
; ModuleID = 'fib.c'
target data_layout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:
  64:64-f32:32:32-f64:64:64-v64:64:64-v128:128:128-a0:0:64-s0:64:64-
  f80:128:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"
@

#define i32 @fib(i32 %n) #0 {
  entry:
  %retval = alloca i32, align 4
  %n.addr = alloca i32, align 4
  store i32 %n, i32* %n.addr, align 4
  %0 = load i32* %n.addr, align 4
  %cmp = icmp eq i32 %0, 0
  br i1 %cmp, label %if.then, label %if.end

  if.then:
  store i32 0, i32* %retval
  br label %return

  if.end:
  %1 = load i32* %n.addr, align 4
  %cmp1 = icmp eq i32 %1, 1
  br i1 %cmp1, label %if.then2, label %if.end3

  if.then2:
  store i32 1, i32* %retval
  br label %return

  if.end3:
  %2 = load i32* %n.addr, align 4
  %sub = sub nsw i32 %2, 1
```
Listing A.1: Sample LLVM IR for fib.c

1 %call = call i32 @fib(i32 %sub)
2 %3 = load i32* %n.addr, align 4
3 %sub4 = sub nsw i32 %3, 2
4 %call5 = call i32 @fib(i32 %sub4)
5 %add = add nsw i32 %call, %call5
6 store i32 %add, i32* %retval
7 br label %return
8
9 return:
10   %if.then2, %if.then
11   %4 = load i32* %retval
12   ret i32 %4
13 }
14
15 ; Function Attrs: nounwind uwtable
16 define i32 @main() #0 {  
17   entry:
18   %retval = alloca i32, align 4
19   %x = alloca i32, align 4
20   store i32 0, i32* %retval
21   %call = call i32 @fib(i32 45)
22   store i32 %call, i32* %x, align 4
23   %0 = load i32* %x, align 4
24   %call1 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8*] @.str, i32 0, i32 0), i32 %0)
25   ret i32 0
26 }
27
28 declare i32 @printf(i8*, ...) #1
29
30 attributes #0 = { nounwind uwtable "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf"="true"  "no-infs-fp-math"="false" "no-nans-fp-math"="false" "stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }
31 attributes #1 = { "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf"="true" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }

Listing A.1: Sample LLVM IR for fib.c
Appendix B

Source Code

B.1 Sample Transformation Using LLVM C++ API

```cpp
bool signaturesMatch(Function *First, Function *Second) {
  if (First == NULL || Second == NULL)
    return false;

  unsigned FirstNumArgs = First->arg_size();
  unsigned SecondNumArgs = Second->arg_size();

  // The number of arguments passed to the old function must
  // match the number of arguments passed to the new function
  if (FirstNumArgs != SecondNumArgs)
    return false;

  // Both functions must abide by the same calling convention
  if (First->getCallingConv() != Second->getCallingConv())
    return false;

  // Both functions must have the same return type
  if (First->getReturnType() != Second->getReturnType())
    return false;

  // Checks that the arguments to the old function are of the same
  // type as those of the new function, and also that they are
  // in the same order
  for (Function::arg_iterator I = Second->arg_begin(), J = First->
    arg_begin(),
    IE = Second->arg_end(); I != IE; ++I, ++J) {
    if (I->getType() != J->getType())
      return false;
  }

  return true;
}

bool replaceFunc(Function *OldFunc, Function *NewFunc) {
```
if (OldFunc == NULL || NewFunc == NULL)
    return false;

if (!signaturesMatch(OldFunc, NewFunc))
    return false;

// Gathers all the calls to the function we want to bypass
InstList Calls = getCallsToFunction(OldFunc);

// Iterates over each call to the function we want to hook
// and sets the callee to the function we want to hook
for (InstList::iterator I = Calls.begin(), E = Calls.end(); I != E; ++I)
{
    CallSite CS(cast<Value>(*I));
    CS.setCalledFunction(NewFunc);
}

// Creates an edge from the calling node to its new destination node
CallGraphNode *CallingNode = (*CG)[CS.getCaller()];
CallGraphNode *NewCalleeNode = (*CG)[NewFunc];
CallingNode->replaceCallEdge(CS, CS, NewCalleeNode);

// Replace all remaining uses of OldFunc with NewFunc
OldFunc->replaceAllUsesWith(NewFunc);

return true;

Listing B.1: C++ source code for implementing the function replacement transformation.
Appendix C

SSH Sessions

C.1 Scripted SSH Session

```bash
#!/usr/bin/expect -f

set IPAddress "[redacted]"
set Username "[redacted]"
set Password "[redacted]"

spawn ssh -o "StrictHostKeyChecking no" $Username@$IPAddress -p 2222
expect "*password:
send "$Password\r"
expect "*Last login:"
send "cd ~/Desktop/\r"
send "rm test.txt\r"
send "ls -la | grep test.txt\r"
send "echo "Copy successful" > test.txt\r"
send "cp test.txt test_copy.txt\r"
send "diff test.txt test_copy.txt\r"
send "cat test_copy.txt\r"
send "logout\r"

Listing C.1: Scripted SSH session script.
```

```bash
[[redacted]@[redacted] Desktop]$ ./regular_session
spawn ssh -o StrictHostKeyChecking no [redacted]@[redacted] -p 2222
[redacted]@[redacted]'s password:
Last login: Tue Mar 18 16:52:28 2014 from [redacted]
[[redacted]@[redacted] Desktop]$ cd ~/Desktop/
[[redacted]@[redacted] Desktop]$ rm test.txt
[[redacted]@[redacted] Desktop]$ ls -la | grep test.txt
[[redacted]@[redacted] Desktop]$ echo "Copy successful" > test.txt
[[redacted]@[redacted] Desktop]$ cp test.txt test_copy.txt
[[redacted]@[redacted] Desktop]$ diff test.txt test_copy.txt
[[redacted]@[redacted] Desktop]$ cat test_copy.txt
```

[[redacted]@[redacted] Desktop]$
C.2 Repeated SFTP Session

Listing C.2: Scripted SSH session output.

Listing C.3: Scripted SFTP session script.
spawf sftp [redacted]@[redacted]
Connecting to [redacted]...

[redacted]@[redacted]'s password:
sftp> lpwd
Local working directory: /export/home/[redacted]/Desktop
sftp> cd Desktop
sftp> !ls -la | grep gettysburg_copy.txt
-rw-rw-r-- 1 [redacted] [redacted] 1463 Mar 18 16:49 gettysburg_copy.txt
sftp> rm gettysburg_copy.txt
Removing /export/home/[redacted]/Desktop/gettysburg_copy.txt
sftp> !ls -la | grep gettysburg_copy.txt
Shell exited with status 1
sftp> put gettysburg.txt gettysburg_copy.txt
Uploading gettysburg.txt to /export/home/[redacted]/Desktop/gettysburg_copy.txt
gettysburg.txt 100% 1463 1.4KB/s 00:00
sftp> !ls -la | grep gettysburg
-rw-rw-r-- 1 [redacted] [redacted] 1463 Mar 18 16:58 gettysburg_copy.txt
-rw-rw-r-- 1 [redacted] [redacted] 1463 Mar 18 15:23 gettysburg.txt
-rw-rw-r-- 1 [redacted] [redacted] 0 Mar 18 15:23 gettysburg.txt
sftp> !diff gettysburg.txt gettysburg_copy.txt
sftp> quit

Listing C.4: Scripted SFTP session output.

C.3 sshd Log

debug1: sshd version OpenSSH_6.4, OpenSSL 1.0.0-fips 29 Mar 2010
debug1: read PEM private key done: type RSA
debug1: private host key: #0 type 1 RSA
debug1: read PEM private key done: type DSA
debug1: private host key: #1 type 2 DSA
debug1: rexec_argv[0]="/home/[redacted]/Desktop/[redacted]/samples/openssh-6.4pl/sshd"
 debug1: rexec_argv[1]="-f"
 debug1: rexec_argv[2]="sshd_config"
 debug1: rexec_argv[3]="-D"
 debug1: rexec_argv[4]="-d"
 debug1: rexec_argv[5]="-e"
 Set /proc/self/oom_score_adj from 0 to -1000
debug1: Bind to port 2222 on 0.0.0.0.
Server listening on 0.0.0.0 port 2222.
socket: Address family not supported by protocol
debug1: Server will not fork when running in debugging mode.
debug1: rexec start in 4 out 4 newsock 4 pipe −1 sock 7
debug1: sshd version OpenSSH_6.4, OpenSSL 1.0.0−fips 29 Mar 2010
debug1: read PEM private key done: type RSA
debug1: private host key: #0 type 1 RSA
debug1: read PEM private key done: type DSA
debug1: private host key: #1 type 2 DSA
debug1: inetd sockets after dupping: 3, 3
Connection from 140.102.30.60 port 45270
debug1: Client protocol version 2.0; client software version OpenSSH_5.3
debug1: match: OpenSSH_5.3 patt OpenSSH_5*
debug1: Enabling compatibility mode for protocol 2.0
debug1: Local version string SSH−2.0−OpenSSH_6.4
debug1: permanently_set_uid: 74/74 [preauth]
debug1: list_hostkey_types: ssh−rsa, ssh−dss [preauth]
debug1: SSH2_MSG_KEXINIT sent [preauth]
debug1: SSH2_MSG_KEXINIT received [preauth]
debug1: kex: client−>server aes128−ctr hmac−md5 none [preauth]
debug1: kex: server−>client aes128−ctr hmac−md5 none [preauth]
debug1: SSH2_MSG_KEX_DH_GEX_REQUEST received [preauth]
debug1: pre_msg_answer_moduli: mm_answer_moduli got sock value 5
WARNING: /usr/local/etc/moduli does not exist, using fixed modulus
debug1: post_msg_answer_moduli: mm_answer_moduli returned 0
debug1: SSH2_MSG_KEX_DH_GEX_GROUP sent [preauth]
debug1: expecting SSH2_MSG_KEX_DH_GEX_INIT [preauth]
debug1: SSH2_MSG_KEX_DH_GEX_REPLY sent [preauth]
debug1: SSH2_MSG_NEWKEYS sent [preauth]
debug1: expecting SSH2_MSG_NEWKEYS [preauth]
debug1: SSH2_MSG_NEWKEYS received [preauth]
debug1: KEX done [preauth]
debug1: userauth−request for user [redacted] service ssh−connection method none [preauth]
debug1: attempt 0 failures 0 [preauth]
debug1: pre_msg_getpwnam: called with username = [redacted]
[preauth]
debug1: post_msg_getpwnam: called with password = *
[preauth]
debug1: userauth−request for user [redacted] service ssh−connection method keyboard−interactive [preauth]
debug1: attempt 1 failures 0 [preauth]
debug1: keyboard−interactive devs [preauth]
debug1: auth2:challenge: user=[redacted] devs= [preauth]
debug1: kbdint_alloc: devices '' [preauth]
debug1: userauth−request for user [redacted] service ssh−connection method password [preauth]
debug1: attempt 2 failures 1 [preauth]
debug1: pre_msg_auth_password: received password = light3mup
[preauth]
Accepted password for [redacted] from 140.102.30.60 port 45270 ssh2
debug1: monitor_child preauth: [redacted] has been authenticated by
  privileged process
debug1: pre_mm_get_keystate: called
debug1: pre_mm_request_receive_expect: mm_request_receive_expect got
  sock value 5
debug1: post_mm_request_receive_expect: called
debug1: post_mm_get_keystate: called
debug1: post_mm_auth_password: mm_auth_password returned 1
  [preauth]
debug1: pre_mm_send_keystate: called
debug1: monitor_read_log: child log fd closed
User child is on pid 21836
debug1: permanently_set_uid: 17102/17102
debug1: Entering interactive session for SSH2.
debug1: server_init_dispatch_20
debug1: server_input_channel_open: ctyype session rchan 0 win 1048576 max
  16384
debug1: input_session_request
debug1: channel 0: new [server-session]
debug1: session_new: session 0
debug1: session_open: channel 0
debug1: session_open: session 0: link with channel 0
debug1: server_input_channel_open: confirm session
debug1: server_input_global_request: rtype no-more-sessions@openssh.com
  want_reply 0
debug1: server_input_channel_req: channel 0 request pty–req reply 1
debug1: session_by_channel: session 0 channel 0
debug1: server_input_channel_req: session 0 req pty–req
debug1: Allocating pty.
debug1: session_new: session 0
debug1: session_pty_req: session 0 alloc /dev/pts/14
debug1: server_input_channel_req: channel 0 request env reply 0
debug1: session_by_channel: session 0 channel 0
debug1: server_input_channel_req: session 0 req env
debug1: server_input_channel_req: channel 0 request shell reply 1
debug1: server_by_channel: session 0 channel 0
debug1: server_input_channel_req: session 0 req shell
debug1: Setting controlling tty using TIOCSCS
debug1: Received SIGCHLD.
debug1: session_by_pid: pid 21837
debug1: session_exit_message: session 0 channel 0 pid 21837
debug1: session_exit_message: release channel 0
debug1: session_by_tty: session 0 tty /dev/pts/14
debug1: session_pty_cleanup: session 0 release /dev/pts/14
debug1: session_by_channel: session 0 channel 0
debug1: session_close_by_channel: channel 0 child 0
debug1: session_close: session 0 pid 0
debug1: channel 0: free: server—session, nchannels 1
Received disconnect from 140.102.30.60: 11: disconnected by user
debug1: do_cleanup
debug1: do_cleanup

Listing C.5: sshd debug output.