Go with the flow: Data Flow Analysis for Binary Differencing

by

Benjamin Denton

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
August 2, 2014

Keywords: Binary Differencing, Reverse Engineering, Data flow Analysis, Static Analysis, Disassembly

Copyright 2014 by Benjamin Denton

Approved by

David Umphress, Chair, Associate Professor of Computer Science and Software Engineering
James Cross, Professor of Computer Science and Software Engineering
Jeffrey Overbey, Assistant Professor of Computer Science and Software Engineering
Michael Hamilton, Assistant Professor of Electrical and Computer Engineering
Abstract

Differencing in computer science is often used to quickly determine differences between two files. While this works well for plain text files, such as source code, applying differencing to binary executable files is more difficult. Compiled binary files contain lists of instructions that when executed, perform operations using functions and data specified by a higher level programming language, such as C++. Syntactic changes to these instructions, changes in the form of an instruction, do not always reflect semantic changes, changes in the behavior of an instruction. Depending on the settings and optimizations of the compiler, a series of instructions from a binary executable could perform the same function as a different series of instructions from a different binary. These types of differences are difficult to detect using current binary differencing methods.

This dissertation explores software reverse engineering, binary differencing, and software semantics vs. syntax. We define a framework, we call Data Flow Binary Differencing, for performing binary differencing using data flow analysis and comparing the semantics of the data flow within a pair of functions. We discuss three use cases that illustrate how to implement the Data Flow Binary Differencing framework and show how our technique stands up against challenges faced by other binary differencing techniques.

Our major contribution of this research is using data flow and assembly language semantics to define a method to compare a pair of functions and test for similarities. We also discover that testing for semantic differences versus syntactic differences within a binary can expose semantic differences introduced by an optimizing compiler.
Acknowledgments

“For the Lord gives wisdom; from his mouth come knowledge and understanding.”

Proverbs 2:6

It cannot be understated how grateful I am to my adviser, Dr. David Umphress, for his patience, support, and encouragement, without which I would not have finished this dissertation. I also would like to thank my committee members, Dr. Overbey, Dr. Cross, and Dr. Hamilton for their time and support of this research. Auburn is an excellent university because of professors like you.

At times this road became lonely and frustrating, I would like to thanks my labmates for providing encouragement, entertainment, and sometimes just an attentive ear to whatever problem I was facing, even though you probably didn’t care. Thank you Devin, Patrick, Chris, Sarah, and Jim for all the laughs and for letting me hang out in your lab. Special thanks to C.W Perr for pushing me through the final months and weeks of this dissertation.

If anyone ever asks me why I did this, I’ll answer to make my Mom and Dad proud. Thank you, Mom and Dad, for all your encouragement and prayer, even when you didn’t understand what I was talking about. I would also like to thank my twin sister, Jeni. You inspire me so much!

The real person that earned this dissertation is my wife, Selena. I just wrote the paper. She has been there through the tears, tantrums, and thwarted attempts to give up. You define me hun! Thanks you for believing in me and always having my back. I love you more everyday!

My most important recognition belong to my Heavenly Father for forgiveness, wisdom, and putting all the people I just mentioned in my life. Thank you God for everything!
Table of Contents

Abstract .................................................. ii
Acknowledgments ......................................... iii
List of Figures ........................................ vii
List of Tables ........................................ viii
1 Introduction ........................................... 1
   1.1 Software Reverse Engineering ..................... 1
      1.1.1 Binary Differencing ............................ 2
      1.1.2 Program Structure ............................. 4
   1.2 Problem Statement ................................ 8
   1.3 Research Summary ................................ 8
2 Literature Survey ................................. 10
   2.1 Symbol Name Matching .......................... 11
   2.2 Signature Hash Map Matching ................... 11
   2.3 Structural Analysis Matching .................. 12
   2.4 Semantic Matching using Symbolic Execution .... 14
   2.5 Summary .......................................... 15
3 Using Data Flow for Function Matching .......... 16
   3.1 A Framework for Data Flow Binary Differencing .. 17
      3.1.1 Step 1: Slicing Instructions .................... 18
      3.1.2 Step 2: Extracting Semantics .................... 19
      3.1.3 Step 3: Testing for Equality ..................... 20
   3.2 Conclusion ..................................... 20
4 Implementing Data Flow Binary Differencing ..... 21
4.1 Research Objective ........................................ 21
4.2 Research Overview ........................................ 22
4.3 Use Case 1: Simple Arithmetic Calculation .............. 23
  4.3.1 Analysis & Observations ............................. 26
  4.3.2 Step 1: Creating Slices ............................. 27
  4.3.3 Step 2: Extracting Semantics ...................... 30
  4.3.4 Step 3: Testing Equality ......................... 31
  4.3.5 Conclusion ....................................... 34
4.4 Use Case 2: Exploring limitations of Data Flow Binary Differencing .......................... 34
  4.4.1 Analysis & Observations ............................. 35
  4.4.2 Step 1: Creating Slices ............................. 40
  4.4.3 Step 2: Extracting Semantics ...................... 43
  4.4.4 Step 3: Testing Equality ......................... 44
  4.4.5 Conclusion ....................................... 49
4.5 Use Case 3: Applying Data Flow Binary Differencing to Actual Malware ............... 49
  4.5.1 Analysis & Observations ............................. 51
  4.5.2 Step 1: Slicing Instructions ...................... 54
  4.5.3 Step 2: Extracting Semantics ...................... 59
  4.5.4 Step 3: Testing for Equality ..................... 60
  4.5.5 Conclusion ....................................... 66
5 Research Conclusion and Future Work .......................... 67
  5.1 Research Contributions ................................. 67
  5.2 Limitations and Challenges ........................... 68
  5.3 Future Work ........................................ 69
Bibliography .................................................. 72
Appendices .................................................. 75
A Source Code ............................................... 76
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Assembly Language</td>
<td>78</td>
</tr>
<tr>
<td>C</td>
<td>Python scripts for unrolling the Fibonacci loop</td>
<td>85</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 Full call graph of helloworld.c .............................................. 6

1.2 Close up of helloworld.c call graph showing _main function .............. 7

1.3 Control flow graph for _main function of helloworld.c ...................... 7

2.1 Overview of the disassembly process ............................................. 10

3.1 Overview of the Data Flow Binary Differencing process ....................... 17

4.1 Control Flow Graph of fib() optimization level 0 .............................. 36

4.2 Control Flow Graph of fib() optimization level 1 .............................. 37

4.3 Control Flow Graph of fib() optimization level 2 .............................. 38

4.4 Control Flow Graph of rndnick() optimization level 0 ......................... 52

4.5 Control Flow Graph of rndnick() optimization level 1 ......................... 53
List of Tables

4.1 Use Case 1: Sources and Sinks for SimpleMath and SimpleMath2 ............ 27
4.2 Structural Analysis of fib() ....................................................... 39
4.3 Use Case 2: Sources and Sinks for fib() optimization level 0 ............... 40
4.4 Use Case 2: Sources and Sinks for fib() optimization level 1 ............... 41
4.5 Use Case 2: Sources and Sinks for fib() optimization level 2 ............... 41
4.6 Structural Analysis of rndnick() ............................................... 51
4.7 Use Case 3: Sources and Sinks for rndnick() optimization level 0 ........... 55
4.8 Use Case 3: Sources and Sinks for rndnick() optimization level 1 ........... 56
4.9 Equations that make Listings 4.33 and 4.38 equal. .............................. 61
5.1 Summary of Binary Differencing Techniques ...................................... 68
### List of Listings

| 4.1 | SimpleMath.c Source Code                                      | 23 |
| 4.2 | SimpleMath2.c Source Code                                     | 23 |
| 4.3 | SimpleMath disassembled by IDA Pro                            | 24 |
| 4.4 | SimpleMath2 disassembled by IDA Pro                           | 25 |
| 4.5 | Difference between source code of Simplemath and SimpleMath2   | 26 |
| 4.6 | Backwards Slicing SimpleMath with source at line 19 and sink at line 23 | 28 |
| 4.7 | Forward Slicing SimpleMath with source at line 19 and sink at line 23 | 29 |
| 4.8 | Backwards Slicing SimpleMath2 with source at line 17 and sink at line 24 | 29 |
| 4.9 | Semantics for Listing 4.6                                     | 30 |
| 4.10| Semantics for Listing 4.7                                     | 30 |
| 4.11| Semantics for Listing 4.8                                     | 30 |
| 4.12| The Z3 Python API prove() function                            | 31 |
| 4.13| Testing slices of SimpleMath and SimpleMath2 for equality      | 32 |
| 4.14| Output of Listing 4.13                                        | 33 |
| 4.15| Backward slice of fib() optimization level 0 from line 17 to line 35 | 42 |
| 4.16| Backward slice of fib() optimization level 1 from line 10 to line 17 | 42 |
| 4.17| Backward slice of fib() optimization level 2 from line 12 to line 21 | 42 |
| 4.18| Backward slice of fib() optimization level 1 from line 24 to line 26 | 43 |
| 4.19| Backward slice of fib() optimization level 2 from line 32 to line 37 | 43 |
| 4.20| Fibonacci array address calculation using optimization levels 0 and 1 | 44 |
| 4.21| Output of Listing 4.20 including Line 32                      | 45 |
| 4.22| Output of Listing 4.20 excluding Line 32                      | 46 |
| 4.23| Fibonacci array address calculation using optimization levels 1 and 2 | 46 |
Chapter 1
Introduction

Software engineering is the process followed by a development team to create well-structured, intuitively designed, and easily maintainable software. There are many different approaches to software engineering, but, in general, each approach includes iterating through phases dedicated to specifying requirements, designing, writing code, testing, and maintaining. The output of each of these phases is a refinement of the previous phase. Software is designed based on requirements, but the design will not necessarily reveal the requirements. The same can be said of the source code and resulting executable binary file: these products do not always reveal the thought put into the requirements and design and in some cases do not even follow the agreed upon requirements and design.

1.1 Software Reverse Engineering

Software reverse engineering takes the reverse approach from the software engineering process. “Reverse engineering is the process of analyzing a subject system to create representations of the system at a higher level of abstraction.”[Chikofsky and Cross 1990] Another way to describe reverse engineering is “the process of extracting the knowledge or design blueprints from anything man-made.”[Eilam 2005] The goal of software reverse engineering is to recover the higher level information that is typically lost in the refinement of the software engineering process. Working typically only with the software executable binary, the software reverse engineer wants to discover the requirements and design of a software product.

The purpose of the software reverse engineering process depends on the goal of the reverse engineering project. In the field of cybersecurity, software reverse engineering is
extremely useful in malware analysis, bug hunting, and software exploitation. Software reverse engineering is also used to re-engineer legacy software where the source code has been lost or is inaccurate. Another important use of software reverse engineering is to implement interoperability between software products. In the absence of source code, software reverse engineering is often the only solution to maintenance and modification of software.

1.1.1 Binary Differencing

Detecting differences in source code, also called ‘diffing’, has been studied extensively [Fluri et al. 2007; Maletic and Collard 2004; Thummalapenta et al. 2009; Kagdi et al. 2007]. Many techniques and tools are available to allow a software engineer to take two versions of the same source code and quickly find the changes between the two versions, such as treating the source code as a sequence of lines and applying a sequence-comparison algorithm [Hirschberg 1977; Hunt and Szymanski 1977; Miller and Myers 1985].

Binary differencing is a much less studied area, since comparing binary files can be difficult. This research focuses on comparing binary executable files. By definition, a binary file is simply a large file of data represented in bytes. A binary executable file contains a list of simple machine instructions that move data between memory and registers, perform mathematical operations, and modify the order of execution of future instructions. These instructions are intermingled with data, so it can be difficult to compare binary executable files directly. A binary analysis tool, called a disassembler, is needed to extract the list of instructions from the other data contain in a binary file.

Once we have a list of instructions, comparing two lists of instructions can also be difficult. Small changes in source code can cause a much larger number of changes in the resulting compiled binary. Understanding where a direct source code change originally occurred can be hidden by these indirect changes. For example, [Gao et al. 2008] reports that changes to 5 lines of source code in a security patch to the gzip application changed all 75 non-empty functions in the resulting binary. Compiler optimizations or changes to local variables in a
source file can effect how registers are allocated in the binary and cause many changes to register usage throughout the binary file. The same is true for memory and data locations in a binary: immediate values and memory addresses are calculated at compile time and changes to source code or compiler settings that shift data or memory locations can change every other memory address in the binary. Each of these changes to registers, immediate values, or addresses in an instruction is easy to detect but do not always map directly to an actual change in the source code.

Binary differencing in software is important to combating the spread of malicious software. Malware often exhibits the same behavior and functionality with small differences between versions and reverse engineering a new version of a piece of malware often requires duplication of effort. A reverse engineer has to re-discover functions and design details already found in the original version. Little effort is required to change source code and recompile. Malware authors take advantage of this problem to evade detection by signature based anti-malware software, which uses a hash or signature of the binary file’s bytes to identify malicious binary files. Malware on Windows and Android platforms are created using modular frameworks that make designing new variations of a single piece of malware easy by providing commonly used functions and utilities. The malware author simply needs to fill in specific details to create unique binaries [Barat et al. 2013; Jiang and Zhou 2013]. Being able to identify these families of malware quickly allows a defender to focus on specific changes between a new variant and a known sample.

Being able to determine changes in software is also important to testing software patches. Proprietary software patch notes often do not fully describe the security problem fixed by the binary patch. Software patches also should be tested prior to deployment to mission critical systems. Using binary differencing techniques, a reverse engineer can determine the changes between the patched and unpatched software and determine if applying the software patch is necessary. Patching software in production systems can be expensive, time consuming, and can run the risk of introducing new software bugs [Cavusoglu et al. 2006]. Binary differencing
can help determine the necessity of a software patch to help justify the expense and to lower the risk of a patch introducing a new bug.

In order to compare binaries, the reverse engineer needs to break the executable binary file into objects suitable for comparison and apply structure to these objects. Using this structure, the reverse engineer can determine which objects exist in both versions and which do not. This is not a simple task since the high-level language constructs and structure are not found in the compiled version of the software.

1.1.2 Program Structure

Programming has evolved from sequential statements with the occasional IF-ELSE and DO constructs to Object-Oriented Design. Program structure is what makes software manageable by humans, whereas computers execute millions of instructions without concern for where they come from or how they are related. Software developers typically break down atomic functionality of a large software product into chunks and implement each chunk using whatever is provided by the software language. Each chunk accomplishes a specific task and can often be developed by different developers or teams of developers. The only requirement of each chunk is that it accomplishes its specific task and communicates well with the other chunks.

The largest chunks that make up a software product are steps, which are collections of independently executable code commonly called libraries. Libraries often represent a specific feature or area of functionality of a program.

The next level of chunks are functions and objects. Functions are sections of code with a well-defined purpose that can be executed from other areas within the program and, optionally, take input from the caller or return data to the caller. Objects extend the idea of functions by limiting code to the data it manipulates. Procedures and objects require data which comes in many different representations such as arrays, linked lists, or trees. Finally, each of these chunks is arranged inside the program using conditional blocks, switch
blocks, and loops that controls the flow of the program. These chunks make developing and understanding the software easier for the development team.

When software is compiled, much of this well thought-out structure can be destroyed by the compiler and linker. Source code files and static system libraries are linked into the same file. Small functions may be inlined, creating one larger chunk out of two smaller chunks. Inefficient code may be optimized by the compiler, such as unrolling a loop where the setup and branching statements cost more than the internal statements. Redundancies within the code may be removed by the compiler to make the resulting binary code smaller or faster. Further, [Balakrishnan and Reps 2010] has shown that the compiler cannot be completely trusted because it and its optimizer tools may subtly, yet significantly, alter the behavior of the program, a phenomenon known as “What you see is not what you execute”. Ultimately, the executable binary file loses much of the structure of the source code and becomes one large chunk of executable statements represented in a machine language.

This transformation from source code to machine language makes software reverse engineering difficult. The software reverse engineer needs to recreate some structure in order to aid in the understanding of the design, functionality, and requirements of the software. Mining meaning from a binary is time consuming without source code and requires the reverse engineer to have knowledge about low-level software, such as assembly language, and computer architecture concepts, such as memory management, in addition to programming knowledge.

To begin creating structure, the executable binary file is disassembled into basic blocks and functions. In general, a basic block is defined as a group of instructions which are always executed together. Functions begin at the target of call statements and end at a retn instruction. In practice, this process can be difficult and is best performed by a disassembler such as IDA Pro [Hex Rays 2014]. The full details of the disassembly process are described further in [Eagle 2011].
Using call graphs and control flow graphs, the disassembly process creates a graph of graphs of a binary that provides the reverse engineer a view that contains functions, loops, and conditional execution paths. Figure 1.1 shows the full call graph of a simple “Hello World!” program. The full call graph is quite large even though the _main function only makes two calls, one to a local function called _getNext() and the other to a library function _printf(), which can be seen in Figure 1.2. Figure 1.3 shows the control flow graph for the _main function. Each node in the call graph represents a control flow graph. These nested graphs are what is meant by graph of graphs. Source code for this program can be found in Appendix A, Listing A.1.

![Figure 1.1: Full call graph of helloworld.c](image)

These graphs, a listing of instructions, and other data provided by the disassembly tool are usually the starting point for a reverse engineering project. This process can require much less work if the reverse engineer has already analyzed a similar binary with similar functions by using binary differencing techniques to detect the functions that have changed between the versions. The goal of binary differencing is to recognize functions and basic blocks that exist in a previous version of the binary, thus cutting down on the effort required to reverse engineer the binary. This allows the reverse engineer to focus on the functions that contain changes in the binary without having to determine which functions have not changed.
Figure 1.2: Close up of helloworld.c call graph showing _main function

Figure 1.3: Control flow graph for _main function of helloworld.c
1.2 Problem Statement

Existing methods to detect changes between different versions of the same binary use static and dynamic techniques. Static techniques include function name matching; matching functions and blocks based on characterizing “signatures”; and structural matching of functions and blocks. These techniques are limited in their ability to discriminate differences in functionality versus differences in form, such as syntactic changes introduced by an optimizing compiler. Such false negatives represent time wasting changes that the reverse engineer must investigate.

Dynamic techniques, on the other hand can detect semantic differences using symbolic execution [Schwartz et al. 2010], but these techniques can be computationally intensive and are typically employed when static techniques have been exhausted. [Gao et al. 2008]

Detecting semantic differences in binaries, as opposed to syntactic differences, is the goal of binary differencing since semantic differences represent a change in functionality. This is a difficult task since syntactic and semantic differences are difficult to tell apart with current binary differencing techniques. Binary differencing techniques need the capability to distinguish between changes that are a direct result of a source code change that modifies program behavior from indirect changes, such as compiler optimizations, that do not affect program behavior.

1.3 Research Summary

This dissertation describes Data Flow Binary Differencing, a technique we developed for use in reverse engineering software. The technique is enabled by data flow analysis over a disassembled binary executable file. By identifying data flows, we are able to translate the series of instructions that represent a data flow into another representation that allows us to compare two data flows from different binary files. If we determine the two data flows are identical, we have shown the binaries that contain these data flows are similar.
The major contribution of this research is the blending of data flow analysis, assembly language semantics, and symbolic execution to create a three step framework for testing functions from separate binary files for similarities. We implement the Data Flow Binary Differencing technique using three use cases of increasing complexity that illustrate the application of the three step framework. For each use case, we create binaries that fail current binary matching techniques by introducing non-semantic differences in the source code or by compiling the source code at different optimization levels. We successfully identify similarities between the binaries from each use case. We also identify semantic differences introduced by an optimizing compiler as limitations of the Data Flow Binary Differencing technique.
Chapter 2
Literature Survey

Binary differencing begins with disassembling a binary file into a series of instructions, basic blocks, and functions. Figure 2.1 shows a simplified overview of this process. A binary file is disassembled by a disassembly tool (in this example IDA Pro) to instructions, then a control flow graph is created from these instructions. The disassembler performs the analysis required to separate instructions into basic block and functions.

Figure 2.1: Overview of the disassembly process
2.1 Symbol Name Matching

The simplest way to match functions between binaries is to match symbol names found in the listing of disassembled instructions. [Wang et al. 2000] demonstrates this by using the symbol names in a binary to match functions with the same symbol name in a similar binary. This can be a fast technique to determine if two binaries contain the same or different functions, but it is dependent on the existence of symbols within a binary. Even if symbols are available in a binary, differences in symbols may indicate only a function name change and does not always reflect a change in the function, thus rendering this approach ineffective.

2.2 Signature Hash Map Matching

To overcome the inability to match functions when symbol names contain small changes, [Wang et al. 2000] uses a hashing-based basic block matching algorithm between basic blocks and functions to create a signature that can be used to check for identical blocks and functions in another binary. [Oh 2009] uses a similar approach by creating a fingerprint hash map of basic blocks, then comparing basic blocks using the computed fingerprints. Both of these techniques depend on creating a unique signature resilient to simple changes such as symbol renaming and register allocation. Signatures are created using instructions inside each basic block, but volatile instruction element, such as address offsets, registers used, and immediate operands, might need to be ignored since they can be changed by the compiler between builds, even without source code changes [Wang et al. 2000].

The signature hash map matching technique depends on the contents of a basic block to detect differences. Source code modifications or compiler or linker optimizations that change instructions such that a different signature is calculated will cause false negatives in the output of this technique when trying to match functions between two binaries. Some of these weaknesses can be overcome, such as instruction reordering implemented by [Oh 2009]. By executing an instruction order normalizing routine in which instructions are ordered by
dependency on each other and sorted by number of bytes, the signature hash map matching technique can increase matching of basic blocks when instructions have simply been reordered inside a basic block. Other weaknesses that do not reflect semantic differences, such as compiler optimizations, cannot be corrected using signature hash map matching.

Signature hash map matching also suffers from collisions when an executable contains short basic blocks. This means identical signatures are calculated over two separate basic blocks within the same executable. The blocks may only differ in elements ignored by the signature calculating algorithm, so they end up having the same signature. [Oh 2009] simply ignores these collisions but suggests that associating signatures with adjacent signatures in the control flow graph, a linked block signature can be checked against colliding block signatures, potentially elimination single block collisions.

2.3 Structural Analysis Matching

Another approach to detect similarities and differences between binaries is to ignore the instructions completely and focus solely on the control flow and call graphs of two binaries as a means to determine similarities between them. Using the call graph, the executable is visualized as a directed graph with functions represented as nodes and the call relationships between functions as edges. The control flow graph for each function is then used to create a directed graph with nodes representing basic blocks and edges representing jump statements. These graphs can be compared for isomorphism, indicating a match between functions.

Comparing undirected graphs can be computationally expensive so [Flake 2004] introduces a simpler method that compares the number of nodes and edges within a graph. This is a fast, although imprecise, technique to test for isomorphism between two graphs. If two graphs contain a different number of nodes and edges, they cannot be isomorphic. This is a quick technique to generate a small subset of potential graph based matched functions for further analysis.
[Flake 2004] and [Dullien and Rolles 2005] both implement a more precise structural analysis matching technique by generating metadata about each function. [Flake 2004] uses a triplet to record the number of basic blocks in a function, number of edges in the function, and number of edges in the call tree starting at the function. For example, the \_main function of helloworld.c in figure 1.3 would have a triplet of \{4,4,2\}, corresponding to its 4 nodes, 4 edges (2 conditional, 2 unconditional), and 2 calls (to \_getNext and \_puts). Functions then are matched according to their metadata represented by this triplet.

[Dullien and Rolles 2005] extends this idea of a triplet by creating what they call selectors. A selector is an algorithm that, given two sets of nodes, returns either a node from each set (indicating a match) or the empty set (no matching nodes found). These two sets of nodes represent basic blocks, functions, and call graphs from the two binaries being tested. Selectors are run iteratively over a collection of nodes until no more matches are made between the two sets. Example of selectors from [Dullien and Rolles 2005] include: \(k\)-indegree/\(k\)-outdegree selector (which selects nodes that contain \(k\) indegree or outdegree edges), recursive node selector (which selects nodes that link to themselves), same name selector (selects nodes with identical symbol name), same string selector (selects nodes that reference the same string), and call graph node selector (selects nodes from the call graph which have the same triplet describe by [Flake 2004]).

Structural analysis matching will not detect small changes to source code such as changing a “less-than” comparison to a “less-than or equal” or the modification of a data type such as changing a local variable from an \texttt{int} to an \texttt{unsigned int}. Either of these changes could represent a security flaw in the application. [Economou 2009] uses structural analysis but also applies heuristics such as instruction counting and calculating a checksum over the basic block, to increase the possibility of detecting small source code changes. Adding introspective analysis to each function and basic block improves structural code analysis techniques but still can falsely identify changes within a function that do not actually change the program behavior.
Another problem for this technique is that small source code changes or compiler optimization might completely rearrange basic blocks, changing the control flow graph. Optimizing compilers can also break apart a single block, adding unconditional jump statements between these sub-blocks. This is especially true of Microsoft’s linker. [Flake 2004] The other code changes are harder to detect without including the actual instructions in the analysis or extending the analysis method.

Structural analysis matching techniques also tend to break down when the binary contains a number of small functions. In general, basic blocks tend to have exactly two children, which represents conditional branches. For a binary with many small functions, multiple function can have the same structure, number of blocks, and call graphs. This makes it difficult for structural analysis matching to distinguish differences between small functions. [Dullien and Rolles 2005]

### 2.4 Semantic Matching using Symbolic Execution

[Gao et al. 2008] implemented symbolic execution and theorem proving as a dynamic binary differencing technique to test basic blocks for semantic differences. This technique can differentiate between syntactic and semantic differences, but due to computation requirements, is only useful for matching individual basic blocks and not functions.

In order to perform symbolic execution, this technique operates on a simple intermediate language. The process takes two basic blocks, identifies the input and output registers, and represents the instructions of each in the intermediate language, which may or may not fully represent the semantics of a machine language. Next, symbolic execution is used to represent the final values of the output registers using the input registers symbols. The two basic block are functionally equivalent if a theorem prover is able to show that the symbolic output values are the same for both. [Gao et al. 2008] never found an example where the simplicity introduced by utilizing an intermediate language hid a semantic change in the
machine language, but the possibility exists. Using symbolic execution to detect semantic differences between binary is a good possibility but can be computationally intensive.

The largest weakness of [Gao et al. 2008]’s method is that it only allows for the comparison of basic blocks. When comparing two functions using this method, the basic block could be determined to be different, when the functions, in fact, may be functionally equivalent. This could be due to compiler optimization that has modified the function’s structure so that the semantics of a basic block in one of the functions is split between two basic block in the other function.

2.5 Summary

In practice, some or all of these techniques are incorporated into tools when used on practical reverse engineering exercises. All static techniques share a weakness of difficult in determining is a change in a binary is a semantic change or a syntactic change. This is often left up to the reverse engineer, who can waste time and resources determining an identified change by one of these methods is actually not a functional change at all.
Chapter 3
Using Data Flow for Function Matching

Considering the limitations of the binary differencing techniques discussed in Chapter 2, our approach shifted the focus from program structure to data structure as a source of comparison. We accomplish this by looking at data flow through a function and comparing the data flow to another function to determine if the two functions are similar. We call this method Data Flow Binary Differencing. This research applies the data flow concepts of dynamic analysis to static analysis so as to identify behavior changes in binaries which may also have structural changes that do not affect behavior. Using data flow analysis to determine function and basic block matching between binaries allows for syntactic differences in implementation, but is able to use the program’s data to determine if any semantic changes occurred between the binaries. Focusing on data flow allows for syntactic differences to exist in the machine languages of two binary executables while preserving the semantics of the intent of the instructions.

We limit the application of our research to comparing two function to determine if they contain similar behavior, regardless of syntactic differences. We assume that, as part of an analysis of a binary executable file we have discovered two functions we believe to contain similarities, but applying the differencing techniques discussed in Chapter 2 suggests the two function are different. Our research begins at this point in the analysis and attempts to prove the two functions contain similarities. Our research does not attempt to replace existing binary differencing methods, but to provide another technique for determining matches between functions.
3.1 A Framework for Data Flow Binary Differencing

Our Data Flow Binary Differencing framework consists of three different steps. Both binaries under comparison are processed in the first two steps and the comparison occurs in the third. The first step identifies slices of instructions within a function, where a slice is a series of instructions. Each slice begins with a data read (load mnemonic) or as the return value of a call instruction, and ends with either a data write (store mnemonic) or at the end of the data flow, the data flow we are following is no longer used. The second step translates the series of instructions to a representation that explicitly represents the semantics. From the semantics, the stream of instructions is represented as a logical formula. In the final step, this logical formula is compared to another logical formula to test for equality. Overall this framework allows for comparison of data usage between binaries with similar functions. We make the assumption that the behavior of a function is exposed by how it interacts with data and we represent this interaction with a logical formula that can be compared to another formula to test for similar behavior.

Figure 3.1: Overview of the Data Flow Binary Differencing process

Data Flow Binary Differencing addresses the limitation of the existing structural, signature, and symbolic analysis methods described in Chapter 2. This technique is not dependent on the control flow graph of a function, a weakness of the Structural Analysis Matching method, nor does this technique depend on syntactic signatures used by the Signature Hash Map Matching technique. Our technique is very similar to the Semantic Matching using...
Symbolic Execution although by focusing on data we are able to generate semantic representations across basic blocks without creating computationally intensive requirements.

### 3.1.1 Step 1: Slicing Instructions

The first step takes the idea of slicing from [Jackson and Rollins 1994], who implement two sets of variable instances, sources and sinks, to create a linear graph showing the instructions that cause the definition of a source to affect the use of a sink. A slice represents the list of instructions that affect the value of the data use at the sink starting at the data definition at the source. The purpose of slicing in our research is to extract from a binary a subset of instructions that define a data flow. We use both forward and backward slicing from [Jackson and Rollins 1994]. Slicing follows the data uses from the source, ending at the sink for forward slicing, and follows data definitions from the sink backward through the instruction flow to the source for backward slicing. Both methods record each instruction that contains a data dependency, generating a series of instruction that represent a data flow.

The primary means of identifying sources and sinks is memory reads and memory writes, respectively defined in our research by a mov instruction with an operand that references a memory location. By slicing between these two points, we obtain the list of instructions that operate on a piece of data. The other primary means of identifying sources and sinks are call instructions. Viewed from the perspective of the calling code, sources can be defined as the return value from a subroutine and sinks can be defined as the arguments to a subroutine.

Using call instructions to identify sources and sinks allows us to model data flow to and from a subroutine. For well known subroutines, such as standard library functions and system calls, we look up the number of arguments so we know which data definitions belong to the slice. We then introduce constraints on the sources and sinks from the call instruction to simplify the matching in the third step. For example, if we know the return value of a standard library function is always ‘1’ or ‘0’, then we can simplify the comparisons
in Step 3 by only considering ‘1’ or ‘0’ as potential values for this variable within a data flow. We also do this for conditional branching conditions. For a data flow that follows a conditional path, we record the condition and, in Step 3, use this as a constraint for the data along this path. An example is a jump condition that is dependent on the value of one symbolic variable being less than another. We include this constraint in our comparison of these data flows which decreases the possible values the symbolic variable can represent.

3.1.2 Step 2: Extracting Semantics

In contrast to high level languages such as C++ and Java, machine languages do not differentiate between data types, such as Int or Bool. Data is simply an \(n\)-bit binary value where \(n\) is determined by the operation of the instructions and width of addressable memory. Instructions perform logical or arithmetic transformations on these \(n\)-bit binary values and move them between memory locations or registers. These logical transformations consist of bitwise operations such as \texttt{and}, \texttt{or}, \texttt{shl} (unsigned left shift), and \texttt{sar} (signed right shift). Arithmetic transformation are mathematic calculations over bit vectors such as \texttt{add}, \texttt{sub}, and \texttt{imul} (integer multiply). The goal of this step is to extract the semantics of the series of instructions created in Step 1. This means explicitly representing the behavior of instructions belonging to complex architectures, where the result of an instruction depends on the contents of the EFLAGS register, the size of the operands, and the mnemonic of the instruction. Each data flow identified in the slicing step is processed in the extracting step.

Some instructions do not contribute to data flow analysis. These instructions can be discarded within the slicing step, but often it is useful to maintain full knowledge of semantics since this could lead to simplification in the comparison step. For example, a \texttt{test} instruction performs a bitwise \texttt{and} on the two operands and modifies the Adjust Flag (AF), Carry Flag (CF), Overflow Flag (OF), Parity Flag (PF), Sign Flag (SF), and Zero Flag (ZF). The CF and OF flags are set to zero. The SF is set to the most significant bit of the result of the \texttt{and} operation. The ZF flag is set to 1 if the result of the \texttt{and} is 0, otherwise the ZF flag is
set to 0. The parity flag is set to 1 if the result of the and operation has an even number of 1 bits. The value of the AF flag is undefined after a test instruction. The result of the and operation is discarded. [Intel Corporation 2014] Subsequent instructions, such as jle (jump less than or equal), use these flags to modify control flow. Using this control flow information we can create constraints for program slices that simplify the comparison between data flows, but we do not need the control flow information to represent the data flow. This is further illustrated in Chapter 4.

3.1.3 Step 3: Testing for Equality

Once we extract the semantics of a series of instructions that represent a single data flow, we test it against data flows in another function we believe contains similar behavior by symbolically representing input to the data flows and testing both flows to see if they produce the same results. We translate the semantics to a single logical expression with symbolic bit vector inputs that evaluates to a single output. We do this for each series of instructions that we would like to test against each other for similarities. We then attempt to prove the logical expressions obtained from the semantics of each series of instruction is equal for the possible values of the symbolic bit vectors. Otherwise, we know the series of instructions contain a semantic difference.

3.2 Conclusion

Our approach to solve the problems identified in Chapter 2 with current binary differencing techniques entails using data flow to characterize the behavior of functions. Using data flow circumvents dependencies on program flow as well as internal structure of a binary or function. We build our research on symbolic execution but focus wholly on data flow as the source of our symbols, allowing us to overcome the computationally requirements of traditional symbolic execution.
Chapter 4
Implementing Data Flow Binary Differencing

4.1 Research Objective

The primary research goal is to determine if syntactic differences between two function represent semantic differences. If we determine that syntactic differences between two functions are not semantic differences, we have confidence that the function are similar, meaning, although the functions look different in their assembly language representation, they perform the same action or actions. We compare the Data Flow Binary Differencing method to the other differencing methods discussed in Chapter 2, specifically Signature Hash Map Matching and Structural Analysis Matching, since these matching techniques perform the same basic action of testing two functions for similarities. We ignore Symbol Name Matching since the matching criteria for this method has nothing to do with the syntax or semantic of a function.

We operate under the assumption that we have two functions that we want to test for differences, ultimately deciding if the functions contain similar behavior. Having some knowledge about the binaries we are testing and an indication that two functions might be equal is the starting point for our research. We do not attempt to solve the complete problem of performing binary differencing without any prior knowledge about the binaries or functions, only to provide another matching technique to test functions that will be used alongside existing techniques. Any difference that cannot be successfully matched semantically between functions is reported to the reverse engineer for further analysis.
4.2 Research Overview

To demonstrate proof of feasibility of our approach, we use IDA Pro ver. 6.6 [Hex Rays 2014] and Python ver. 2.7.3 to execute the first two steps of the Data Flow Binary Differencing Framework and then use the Z3 SMT solver ver. 4.3.2 [Microsoft Research 2014] to test for equality in the third step. From the available tools, this approach provided the greatest flexibility while meeting our research goals.

The process begins by compiling the use cases using GCC 4.7.2 on Debian 7 (Linux kernel 3.2.0-4-amd64). Once we have creating sample binaries, we use IDA Pro to disassemble the binaries to assembly language instructions. After disassembly, IDA performs a number of static analyses on the instructions including identifying local variables, auto-generating and populating names for known symbols, and creating a graph view of basic blocks and control flow of the instructions. Using this information, we create slices of instructions based on the method discussed in Section 3.1.1. Next, we extract the semantics of each instruction contained within each slice and explicitly represent the behavior of each instruction using Python. We test the two outputs for equality using Z3, a software solver for SMT (Satisfiability Modulo Theories), an extension of the Satisfiability (SAT) problem.

The assembly language listing and control flow graph figures presented in this dissertation are generated by IDA Pro. For local variables located on the stack, IDA Pro assigns a label to that variable relative to the position of the variable on the stack. For example, a variable located at the offset -Ch from the base of the stack, usually the value of the ebp register, would be referenced in IDA Pro as [ebp+var_C]. To represent hexadecimal in this dissertation we follow the format used by IDA Pro and represent hexadecimal values with “h” and hexadecimal addresses with “0x”, such as -Ch is a value and [0x080486A3] is an address.

The following sections describe three use cases, each illustrating increasing complexity and achieving slightly different research objectives. These uses cases were selected to cover the three fundamental program flow constructs found in programming languages, sequential execution, conditional execution, and looping.
4.3 Use Case 1: Simple Arithmetic Calculation

Our first use case illustrates the approach using a simple mathematic example and allows us to show how the Data Flow Binary Differencing technique is applied. We create two functions that add two numbers together. The second function adds these two numbers in a different order and has an additional addition and subtraction operation but ultimately arrives at the same result as the first function. The goal of this use case is to show that our technique can be applied to syntactically different functions with semantically equal functionality.

Listings 4.1 and 4.2 are compiled using gcc, resulting in the 32-bit assembly language shown in Listings 4.3 and 4.4.

Listing 4.1: SimpleMath.c Source Code

```c
#include <stdlib.h>

int main(int argc, char *argv[]){
    int w, x, y, z;
    x = 2;
    y = 3;
    w = 5;
    z = x + y;
    return z;
}
```

Listing 4.2: SimpleMath2.c Source Code

```c
#include <stdlib.h>

int main(int argc, char *argv[]){
    int x, y, z;
    x = 2;
    y = 3;
    z = y + x + x - x;
    return z;
}
```
Listing 4.3: SimpleMath disassembled by IDA Pro

```assembly
; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

var_10 = dword ptr -10h
var_C = dword ptr -0Ch
var_8 = dword ptr -8
var_4 = dword ptr -4
argc = dword ptr 8
argv = dword ptr 0Ch
envp = dword ptr 10h

push ebp
mov ebp, esp
sub esp, 10h
mov [ebp + var_4], 2
mov [ebp + var_8], 3
mov [ebp + var_C], 5
mov eax, [ebp + var_8]
mov edx, [ebp + var_4]
add eax, edx
mov [ebp + var_10], eax
mov eax, [ebp + var_10]
leave
retn
main endp
```
```
Listing 4.4: SimpleMath2 disassembled by IDA Pro

<table>
<thead>
<tr>
<th></th>
<th>int __cdecl main(int argc, const char **argv, const char **envp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>; int __cdecl main(int argc, const char **argv, const char **envp)</td>
</tr>
<tr>
<td>2</td>
<td>public main</td>
</tr>
<tr>
<td>3</td>
<td>main proc near</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>var_C = dword ptr -0Ch</td>
</tr>
<tr>
<td>6</td>
<td>var_8 = dword ptr -8</td>
</tr>
<tr>
<td>7</td>
<td>var_4 = dword ptr -4</td>
</tr>
<tr>
<td>8</td>
<td>argc = dword ptr 8</td>
</tr>
<tr>
<td>9</td>
<td>argv = dword ptr 0Ch</td>
</tr>
<tr>
<td>10</td>
<td>envp = dword ptr 10h</td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>push ebp</td>
</tr>
<tr>
<td>13</td>
<td>mov ebp, esp</td>
</tr>
<tr>
<td>14</td>
<td>sub esp, 10h</td>
</tr>
<tr>
<td>15</td>
<td>mov [ebp+var_4], 2</td>
</tr>
<tr>
<td>16</td>
<td>mov [ebp+var_8], 3</td>
</tr>
<tr>
<td>17</td>
<td>mov eax, [ebp+var_4]</td>
</tr>
<tr>
<td>18</td>
<td>mov edx, [ebp+var_8]</td>
</tr>
<tr>
<td>19</td>
<td>add edx, eax</td>
</tr>
<tr>
<td>20</td>
<td>mov eax, [ebp+var_4]</td>
</tr>
<tr>
<td>21</td>
<td>add eax, edx</td>
</tr>
<tr>
<td>22</td>
<td>sub eax, [ebp+var_4]</td>
</tr>
<tr>
<td>23</td>
<td>mov [ebp+var_C], eax</td>
</tr>
<tr>
<td>24</td>
<td>mov eax, [ebp+var_C]</td>
</tr>
<tr>
<td>25</td>
<td>leave</td>
</tr>
<tr>
<td>26</td>
<td>retn</td>
</tr>
<tr>
<td>27</td>
<td>main endp</td>
</tr>
</tbody>
</table>
```
4.3.1 Analysis & Observations

The source code of SimpleMath and SimpleMath2 contain minor syntactic differences shown using the `diff` command in Listing 4.5. Since this function contains only one basic block, the Structural Analysis Matching technique alone attempts to match this function to any other function with a single basic block. The Signature Hash Map Matching technique fails because the additional instructions in SimpleMath2 will cause the Signature Hash Map Matching technique to generate a different signature than SimpleMath.

<table>
<thead>
<tr>
<th>Listing 4.5: Difference between source code of SimpleMath and SimpleMath2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>diff SimpleMath.c SimpleMath2.c</code></td>
</tr>
<tr>
<td>4c4</td>
</tr>
<tr>
<td><code>&lt; int w, x, y, z;</code></td>
</tr>
<tr>
<td><code>&gt;</code></td>
</tr>
<tr>
<td><code>int x, y, z;</code></td>
</tr>
<tr>
<td>7,8c7</td>
</tr>
<tr>
<td><code>&lt; w = 5;</code></td>
</tr>
<tr>
<td><code>&lt; z = x + y;</code></td>
</tr>
<tr>
<td><code>&gt;</code></td>
</tr>
<tr>
<td><code>z = y + x + x - x;</code></td>
</tr>
</tbody>
</table>

SimpleMath includes an additional local variable declaration on line 4, `int w`, and an additional assignment on line 7 of Listing 4.1, `w = 5`. IDA Pro identifies the additional local variable `w` in SimpleMath and assigns it the label `var_C` (Listing 4.3 line 6 and 18). We can see from the assembly in Listing 4.3, after the definition of `var_C` on line 18 the data is never used so this variable does not contribute to a data flow.

SimpleMath adds the local variables `x` and `y` on line 8 of Listing 4.1. SimpleMath2 does the same but also performs an additional addition and subtraction on line 7 of Listing 4.2. The value assigned to the local variable `z` is the same as the value returned by the `main` function in both programs despite the additional mathematic operations on line 7 of SimpleMath2. We know that, semantically, `x + y == y + x + x - x` is true. Our technique will prove the semantic equality of these calculation by proving these output values are equal.
4.3.2 Step 1: Creating Slices

Sources are defined as memory load operations. Lines 19, 20, and 23 in SimpleMath\textsuperscript{1} read data from memory (the stack in this case) and load that value into a register. Lines 17, 18, 19, and 24 in SimpleMath\textsuperscript{2} also read data from memory. Sinks are defined as memory store instructions and as the return value of the function (the value in \texttt{eax} at \texttt{retn}). Also depending on the memory usage of the function, the return value could also be a source, as it is in this case. Data is written to the stack at lines 16, 17, 18, and 22 in SimpleMath\textsuperscript{1} and lines 15, 16, and 23 in SimpleMath\textsuperscript{2} so we consider these instructions as sinks. The return value of both functions is also a sink, line 23 in SimpleMath\textsuperscript{1} and line 24 in SimpleMath\textsuperscript{2}.

Table 4.1 lists all sources and sink in both programs’ main() functions.

<table>
<thead>
<tr>
<th>File</th>
<th>Line</th>
<th>Type</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>SimpleMath</td>
<td>19</td>
<td>Source</td>
<td>mov eax, [ebp+var_8]</td>
</tr>
<tr>
<td>SimpleMath</td>
<td>20</td>
<td>Source</td>
<td>mov edx, [ebp+var_4]</td>
</tr>
<tr>
<td>SimpleMath</td>
<td>23</td>
<td>Source</td>
<td>mov eax, [ebp+var_10]</td>
</tr>
<tr>
<td>SimpleMath</td>
<td>16</td>
<td>Sink</td>
<td>mov [ebp+var_4], 2</td>
</tr>
<tr>
<td>SimpleMath</td>
<td>17</td>
<td>Sink</td>
<td>mov [ebp+var_8], 3</td>
</tr>
<tr>
<td>SimpleMath</td>
<td>18</td>
<td>Sink</td>
<td>mov [ebp+var_C], 5</td>
</tr>
<tr>
<td>SimpleMath</td>
<td>22</td>
<td>Sink</td>
<td>mov [ebp+var_10], eax</td>
</tr>
<tr>
<td>SimpleMath</td>
<td>23</td>
<td>Sink</td>
<td>mov eax, [ebp+var_10]</td>
</tr>
<tr>
<td>SimpleMath2</td>
<td>17</td>
<td>Source</td>
<td>mov eax, [ebp+var_4]</td>
</tr>
<tr>
<td>SimpleMath2</td>
<td>18</td>
<td>Source</td>
<td>mov edx, [ebp+var_8]</td>
</tr>
<tr>
<td>SimpleMath2</td>
<td>20</td>
<td>Source</td>
<td>mov eax, [ebp+var_4]</td>
</tr>
<tr>
<td>SimpleMath2</td>
<td>24</td>
<td>Source</td>
<td>mov eax, [ebp+var_C]</td>
</tr>
<tr>
<td>SimpleMath2</td>
<td>15</td>
<td>Sink</td>
<td>mov [ebp+var_4], 2</td>
</tr>
<tr>
<td>SimpleMath2</td>
<td>16</td>
<td>Sink</td>
<td>mov [ebp+var_8], 3</td>
</tr>
<tr>
<td>SimpleMath2</td>
<td>23</td>
<td>Sink</td>
<td>mov [ebp+var_C], eax</td>
</tr>
<tr>
<td>SimpleMath2</td>
<td>24</td>
<td>Sink</td>
<td>mov eax, [ebp+var_C]</td>
</tr>
</tbody>
</table>

We can see from Table 4.1, a few sinks are found at higher line numbers than any of the identified sources. Without a prior source, we cannot create a slice that includes these

\textsuperscript{1}Line numbers reference Listings 4.3 and 4.4
sinks, so we must ignore them. We also ignore the the last source in both functions since no corresponding sink exists.

We use both forward and backward slicing to calculate slices. Listings 4.6 and Listings 4.7 illustrate the difference between forward and backward slicing.

To calculate a backward slice, we look for all data dependencies between the source and sink that influence the value at the sink. Conceptually, we start at the sink and follow use-define chains backward through the instruction flow until we reach the source. Starting at the sink in Listing 4.6, the instruction mov eax, [ebp+var_10] defines eax using [ebp+var_10]. So we look backwards through the instruction flow for a definition of [ebp+var_10] and add the instruction to the slice. The previous instruction, mov [ebp+var_10], eax, is added to the slice because it defines [ebp+var_10] and uses eax. Now we look for instructions that define eax. The add eax, edx instruction defines eax and uses eax and edx, so we now look for definitions of eax and edx. The next two previous instructions mov eax, [ebp+var_8] and mov edx, [ebp+var_4] define eax and edx, so they are also added to the slice. The instruction mov eax, [ebp+var_8] is also the source so we have finished calculating this slice.

Listing 4.6: Backwards Slicing SimpleMath with source at line 19 and sink at line 23

```
1 mov   eax, [ebp+var_8]
2 mov   edx, [ebp+var_4]
3 add   eax, edx
4 mov   [ebp+var_10], eax
5 mov   eax, [ebp+var_10]
```

Calculating a forward slice is similar, but follows define-use chains from the source to the sink. The source instruction on line 19, mov eax, [ebp+var_8], defines eax using the data from [ebp+var_8], so we look for uses of eax in the following instructions. The next instruction, mov edx, [ebp+var_4] is not added to the slice because it does not use eax. The add eax, edx instruction uses eax, so it is added to the slice. The add eax, edx
redefines eax so we continue looking for uses of eax until we reach the sink. The remaining instructions including the sink are part of the use-define chain with add eax, edx, so they are also added to the slice. The final resulting slice is shown in Listing 4.7. The slice obtained from forward slicing is missing an instruction included in the backward slice, Listing 4.6.

Listing 4.7: Forward Slicing SimpleMath with source at line 19 and sink at line 23

```assembly
mov eax, [ebp+var_8]
add eax, edx
mov [ebp+var_10], eax
mov eax, [ebp+var_10]
```

Our method allows for the creation of forward or backward slices. In Steps 2 and 3 of our framework, we test if either slicing method matches a slice from another binary, with a successful match representing a semantic match between data flows.

Continuing with our process, we calculate a backward slice from SimpleMath2, Listing 4.4. The resulting slice is show in Listing 4.8.

Listing 4.8: Backwards Slicing SimpleMath2 with source at line 17 and sink at line 24

```assembly
mov eax, [ebp+var_4]
mov edx, [ebp+var_8]
add edx, eax
mov eax, [ebp+var_4]
add eax, edx
sub eax, [ebp+var_4]
mov [ebp+var_C], eax
mov eax, [ebp+var_C]
```

Looking back at Table 4.1, we see that the slice from SimpleMath, shown in Listing 4.6, contains all possible sources and sinks in the function, so any other slice generated from this table would be a subset of this slice. The same is true for the slice in Listing 4.8 from SimpleMath2.
4.3.3 Step 2: Extracting Semantics

We extract the behavior of each slice in preparation for symbolic execution, modeling the behavior in Python. Listings 4.9, 4.10, and 4.11 contain the results of extracting semantics for each slice in SimpleMath and SimpleMath2. Actual instructions appear as comments in the Python code for reference.

Listing 4.9: Semantics for Listing 4.6

```
1   eax = var_8    #mov  eax, [ebp+var_8]
2   edx = var_4    #mov  edx, [ebp+var_4]
3   eax = eax + edx #add  eax, edx
4   var_10 = eax   #mov  [ebp+var_10], eax
5   eax = var_10   #mov  eax, [ebp+var_10]
```

Listing 4.10: Semantics for Listing 4.7

```
1   eax = var_8    #mov  eax, [ebp+var_8]
2   eax = eax + edx #add  eax, edx
3   var_10 = eax   #mov  [ebp+var_10], eax
4   eax = var_10   #mov  eax, [ebp+var_10]
```

Listing 4.11: Semantics for Listing 4.8

```
1   eax = var_4    #mov  eax, [ebp+var_4]
2   edx = var_8    #mov  edx, [ebp+var_8]
3   edx = edx + eax #add  edx, eax
4   eax = var_4    #mov  eax, [ebp+var_4]
5   eax = eax + edx #add  eax, edx
6   eax = eax - var_4 #sub  eax, [ebp+var_4]
7   var_C = eax    #mov  [ebp+var_C], eax
8   eax = var_C    #mov  eax, [ebp+var_C]
```

Extracting semantics was performed manually by translating each instruction to the equivalent Python expression. [Intel Corporation 2014] We chose Python as the intermediate language is it allows us to use the Python API for the Z3 solver, discussed in the next step.
Other tools are available that implement an intermediate language such as S2E [Chipounov et al. 2011] and BAP [Brumley et al. 2011], but none of these tools are designed to allow analysis over multiple binary files at the same time.

All of our use cases employ x86 machine language instructions that could be modeled explicitly in the Python programming language.

4.3.4 Step 3: Testing Equality

After obtaining the semantics for each slice, we symbolically execute the slices and test if it matches the behavior of another slice. We implement this using the Microsoft Z3 solver’s Python API. We use the `prove()` function from Z3’s Python API, which creates an assert statement that attempts to disprove the equality of the two input statements. The `prove()` function takes a single input argument, for example `prove (P)`, and tries to solve the negation of that equation, `Not(P) == True`. If the solver cannot prove `Not(P)`, then it has indirectly proven `P == True`.

Listing 4.12 shows a simple example using the `prove` function from the Z3 Python API. Here we create two symbolic bit vectors, both 32-bit in length, and instruct `prove` to test if they are equal. Since both values are symbolic, Z3 is free to chose any value for each value. We do not provide any constraints for the variables so Z3 chooses different values which invalidates the statement `a == b`.

<table>
<thead>
<tr>
<th>Python 2.7.3 (default, Mar 13 2014, 11:03:55)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[GCC 4.7.2] on linux2</td>
</tr>
<tr>
<td>Type &quot;help&quot;, &quot;copyright&quot;, &quot;credits&quot; or &quot;license&quot; for more information.</td>
</tr>
<tr>
<td>&gt;&gt;&gt; from z3 import *</td>
</tr>
<tr>
<td>&gt;&gt;&gt; a = BitVec(&quot;a&quot;,32)</td>
</tr>
<tr>
<td>&gt;&gt;&gt; b = BitVec(&quot;b&quot;,32)</td>
</tr>
<tr>
<td>&gt;&gt;&gt; prove (a == b)</td>
</tr>
<tr>
<td>counterexample</td>
</tr>
<tr>
<td>[b = 0, a = 4294967295]</td>
</tr>
</tbody>
</table>

Listing 4.12: The Z3 Python API `prove()` function
For our use cases, we pass `prove()` a statement representing testing two slices for equality, `prove(slice1 == slice2)`. `prove()` tells us that it has “proven” the statement (our slices are equal) or it provides a counterexample showing us the symbolic values in the counterexample that invalidate our statement.

Our goal is to prove that slices in SimpleMath are equal to slices from SimpleMath2. We compare both Listing 4.6 and 4.7 to Listing 4.8. We test both the forward and backward slices calculated from SimpleMath to the backward slice calculated from SimpleMath2. The complete Python script that implements these tests is shown in Listing 4.13. Running this script produces the output in Listing 4.14.


```python
from z3 import *
s = Solver()
var_4 = BitVec("var_4", 32)
var_8 = BitVec("var_8", 32)

# Semantics of backward slicing SimpleMath from line 19 to line 23
eax = var_8  #mov eax, [ebp+var_8]
edx = var_4  #mov edx, [ebp+var_4]
eax = eax + edx  #add eax, edx
var_10 = eax  #mov [ebp+var_10], eax
eax = var_10  #mov eax, [ebp+var_10]
sm_slice1 = eax

# Semantics of forward slicing SimpleMath from line 19 to line 23
# We don't know the value of eax, so it is replaced with a symbol
edx = BitVec("edx", 32)
eax = var_8  #mov eax, [ebp+var_8]
eax = eax + edx  #add eax, edx
var_10 = eax  #mov [ebp+var_10], eax
eax = var_10  #mov eax, [ebp+var_10]
sm_slice2 = eax

# Semantics of backward slicing SimpleMath2 from line 17 to line 24
eax = var_4  #mov eax, [ebp+var_4]
edx = var_8  #mov edx, [ebp+var_8]
edx = edx + eax  #add edx, eax
eax = var_4  #mov eax, [ebp+var_4]
eax = eax + edx  #add eax, edx
eax = eax - var_4  #sub eax, [ebp+var_4]
var_C = eax  #mov [ebp+var_C], eax
```

32
From lines 4 and 5 of Listing 4.14, we see that the Z3 solver is able to prove the backward slice from SimpleMath, \( sm\_slice1 \) in Listing 4.14, matches the backward slice from SimpleMath2, \( sm2\_slice1 \) in Listing 4.14. The symbolic representation of \( sm\_slice1 \) is \( \text{var}_8 + \text{var}_4 \) and the symbolic representation of \( sm2\_slice1 \) is \( \text{var}_4 + \text{var}_8 + \text{var}_4 - \text{var}_4 \). Z3 is able to prove \( \text{var}_8 + \text{var}_4 = \text{var}_4 + \text{var}_8 + \text{var}_4 - \text{var}_4 \).

Lines 6 and 7 of Listing 4.14 show the result of comparing the forward slice from SimpleMath to the backward slice from SimpleMath2. The solver is unable to prove these two slices are equal and provides a counterexample. Symbolically, the solver tries to prove \( \text{eax} + \text{var}_4 = \text{var}_4 + \text{var}_8 + \text{var}_4 - \text{var}_4 \). The left side of this equation has an extra variable not present on the right side, \( \text{eax} \). The solver returns values for \( \text{var}_8 \) and \( \text{eax} \) that invalidate this equation for all values of \( \text{var}_4 \).
4.3.5 Conclusion

The SimpleMath use case illustrates each step of the Data Flow Binary Differencing framework by creating slices of instructions based on sinks and sources, extracting the semantics from these instructions, and then symbolically testing slices from both programs for equality. When creating slices, forward and backward techniques do not always produce the same slice, so we should test both methods when calculating slices. By comparing the backward slice from SimpleMath and the backward slice from SimpleMath2, we have shown the Data Flow Binary Differencing method for detecting semantically equal, yet syntactically different, functions is successful with this simple example.

4.4 Use Case 2: Exploring limitations of Data Flow Binary Differencing

Use Case 2 demonstrates the validity of our approach on artifacts that contain conditional program flow in the form of loops. This use case is designed to test the limitations of Binary Data Flow Differencing, specifically to see how our approach handles looping. We know that symbolic execution for program analysis tends to have trouble with loops. This was a major limitation of using S2E [Chipounov et al. 2011] or BAP [Brumley et al. 2011] for this research because both implementations require the intermediate language to unroll loops before invoking the SMT solver.

We chose to illustrate the use case with a program that calculates Fibonacci numbers, creating different versions by compiling it with different optimization levels. The use of different optimization levels introduces syntactic differences without modifying the semantics of the program.

GCC supports 4 levels of optimizations. Level 0 instructs the compiler to generate machine instructions without reordering statements or enabling optimizations. At level 1, the compiler tries to reduce code size and execution time without performing any optimizations that increase compilation time. Level 2 includes all the optimizations at level 1 plus optimizations that do not involve a space-speed tradeoff, so the resulting executable will not
increase in size. Level 3 optimizes beyond level 2 by turning on more expensive optimization that can increase compile time and the resulting file size. We used these optimization levels to created semantically identical binaries with syntactic differences.

The source code for our Fibonacci program can be found in Appendix A, Listing A.2. As with our first use case, we build Fibonacci with GCC 4.7.2 on Debian 7 using all four of the optimization levels. Our Fibonacci function takes one argument, \( n \), calculates an array of Fibonacci values, and returns the requested value, the \( n \)-th Fibonacci number. The \( \text{fib()} \) function calculates the array of Fibonacci values by setting the first and second Fibonacci numbers to 0 and 1, respectively, and then loops over an addition operation that adds the previous two Fibonacci numbers and stores the result in the next array location.

We use optimization levels 0, 1, and 2 for our analysis. The binary resulting from optimization level 3 was almost identical to level 2 so we exclude it from the analysis. After compiling the source code at each of the optimization levels 0, 1, and 2, we use IDA Pro to disassemble the resulting binaries. In Appendix B we show the disassembled \( \text{fib()} \) functions from each optimization level. Each binary contains a \( \text{main()} \) function as well as other imported standard library functions not shown, since these additional functions are not relevant to our research.

Figures 4.1, 4.2, and 4.3 show the Control Flow Graph (CFG) of \( \text{fib()} \) at each optimization level 0, 1, and 2.

4.4.1 Analysis & Observations

In looking at the conventional function matching techniques, the Structural Analysis technique depends on similarities between the CFGs to detect similar functions. Table 4.2 shows a way to perform structural analysis on \( \text{fib()} \). The matching algorithm attempts to match functions that contain the closest number of basic blocks, edges, and calls. None of the functions under analysis from the three optimization levels contain the same number of basic blocks, edges, or calls so Structural Analysis matching fails.
Figure 4.1: Control Flow Graph of fib() optimization level 0
Figure 4.2: Control Flow Graph of fib() optimization level 1
Figure 4.3: Control Flow Graph of fib() optimization level 2
Table 4.2: Structural Analysis of fib()

<table>
<thead>
<tr>
<th>Optimization Level</th>
<th>Number of Blocks</th>
<th>Edges</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>-O0</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>-O1</td>
<td>4</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>-O2</td>
<td>5</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

The Signature Hash Map matching technique attempts to create signatures of each basic block using the mnemonics and operands of the instructions. None of the basic blocks in Figures 4.1, 4.2, and 4.3 contain identical instructions so each basic block will have a different signature. Figures 4.2 and 4.3 both contain a basic block with exactly two `mov` instructions, but with different operands. Ignoring the operands, Signature Hash Map Matching could generate a signature that matches basic blocks containing two `mov` instructions, but every other basic block has a different signature so ultimately this matching technique could only match one of four or five basic blocks between these functions.

From a reverse engineering perspective, the `fib()` function is complicated. Each binary is compiled using the exact same source code but with different compiler optimization enabled, the disassembled instructions contain many syntactic differences that make understanding the behavior of each binary difficult.

As for the behavior of the functions, the first operation that occurs is the calculation of the array size and address to store the Fibonacci numbers. The number of Fibonacci numbers to calculate is passed to the function as `arg_0`. The function uses this value to calculate the number of bytes required to hold the Fibonacci array and subtracts that number from the current stack address to create space to store the array.

Optimization level 0 results in a complicated approach to calculate the Fibonacci array shown in Listing B.1, lines 17-35 whereas the other optimization levels use a much simpler approach to calculate the array shown in Listing B.2, lines 10-15 for optimization level 1 and Listing B.3, lines 12-18 for optimization level 2.
4.4.2 Step 1: Creating Slices

Tables 4.3, 4.4 and 4.5 show the sources and sinks from each binary. Without optimizations, the binary includes many memory operations. Optimization levels 1 and 2 have fewer memory load and stores that optimization level 0, which makes runtime faster.

Table 4.3: Use Case 2: Sources and Sinks for fib() optimization level 0

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Type</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>Source</td>
<td>mov eax, [ebp+arg_0]</td>
</tr>
<tr>
<td>36</td>
<td>Source</td>
<td>mov eax, [ebp+var_14]</td>
</tr>
<tr>
<td>38</td>
<td>Source</td>
<td>mov eax, [ebp+var_14]</td>
</tr>
<tr>
<td>44</td>
<td>Source</td>
<td>mov eax, [ebp+var_C]</td>
</tr>
<tr>
<td>46</td>
<td>Source</td>
<td>mov eax, [ebp+var_14]</td>
</tr>
<tr>
<td>47</td>
<td>Source</td>
<td>mov edx, [eax+edx*4]</td>
</tr>
<tr>
<td>48</td>
<td>Source</td>
<td>mov eax, [ebp+var_C]</td>
</tr>
<tr>
<td>50</td>
<td>Source</td>
<td>mov eax, [ebp+var_14]</td>
</tr>
<tr>
<td>51</td>
<td>Source</td>
<td>mov eax, [eax+edx*4]</td>
</tr>
<tr>
<td>53</td>
<td>Source</td>
<td>mov edx, [ebp+var_14]</td>
</tr>
<tr>
<td>54</td>
<td>Source</td>
<td>mov edx, [ebp+var_C]</td>
</tr>
<tr>
<td>58</td>
<td>Source</td>
<td>mov eax, [ebp+var_C]</td>
</tr>
<tr>
<td>61</td>
<td>Source</td>
<td>mov eax, [ebp+var_14]</td>
</tr>
<tr>
<td>62</td>
<td>Source</td>
<td>mov edx, [ebp+arg_0]</td>
</tr>
<tr>
<td>63</td>
<td>Source</td>
<td>mov eax, [eax+edx*4]</td>
</tr>
<tr>
<td>65</td>
<td>Source</td>
<td>mov ebx, [ebp+var_4]</td>
</tr>
<tr>
<td>20</td>
<td>Sink</td>
<td>mov [ebp+var_10], edx</td>
</tr>
<tr>
<td>26</td>
<td>Sink</td>
<td>mov [ebp+var_1C], 0x10</td>
</tr>
<tr>
<td>35</td>
<td>Sink</td>
<td>mov [ebp+var_14], eax</td>
</tr>
<tr>
<td>37</td>
<td>Sink</td>
<td>mov dword ptr [eax], 0</td>
</tr>
<tr>
<td>41</td>
<td>Sink</td>
<td>mov dword ptr [eax+4], 1</td>
</tr>
<tr>
<td>40</td>
<td>Sink</td>
<td>mov [ebp+var_C], 2</td>
</tr>
<tr>
<td>55</td>
<td>Sink</td>
<td>mov [eax+edx*4], ebx</td>
</tr>
<tr>
<td>63</td>
<td>Sink</td>
<td>mov eax, [eax+edx*4]</td>
</tr>
</tbody>
</table>

There are many possible individual slices that can be calculated from Table 4.3 given the large number of sources and sinks. A much smaller number of slices can be calculated from the other two binaries. The slices that were relavent to this research were slices that captured the behavior of the array size calculation and the loop. Listings 4.15, 4.16, and 4.19 show the slices that capture the array size and address calculation for each binary.
Table 4.4: Use Case 2: Sources and Sinks for fib() optimization level 1

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Type</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Source</td>
<td>mov ebx, [ebp+arg_0]</td>
</tr>
<tr>
<td>24</td>
<td>Source</td>
<td>mov ecx, [eax+4]</td>
</tr>
<tr>
<td>32</td>
<td>Source</td>
<td>mov eax, [esi+ebx*4]</td>
</tr>
<tr>
<td>17</td>
<td>Sink</td>
<td>mov dword ptr ds:0[eax*4], 0</td>
</tr>
<tr>
<td>18</td>
<td>Sink</td>
<td>mov dword ptr ds:4[eax*4], 1</td>
</tr>
<tr>
<td>26</td>
<td>Sink</td>
<td>mov [eax+8], ecx</td>
</tr>
<tr>
<td>32</td>
<td>Sink</td>
<td>mov eax, [esi+ebx*4]</td>
</tr>
</tbody>
</table>

Table 4.5: Use Case 2: Sources and Sinks for fib() optimization level 2

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Type</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Source</td>
<td>mov esi, [ebp+arg_0]</td>
</tr>
<tr>
<td>32</td>
<td>Source</td>
<td>mov ecx, [eax+4]</td>
</tr>
<tr>
<td>33</td>
<td>Source</td>
<td>mov ebx, [eax]</td>
</tr>
<tr>
<td>42</td>
<td>Source</td>
<td>mov edx, [ebp+arg_0]</td>
</tr>
<tr>
<td>51</td>
<td>Source</td>
<td>mov eax, [edi+edx*4]</td>
</tr>
<tr>
<td>21</td>
<td>Sink</td>
<td>mov dword ptr ds:0[eax*4], 0</td>
</tr>
<tr>
<td>22</td>
<td>Sink</td>
<td>mov dword ptr ds:4[eax*4], 1</td>
</tr>
<tr>
<td>37</td>
<td>Sink</td>
<td>mov [eax+8], ecx</td>
</tr>
<tr>
<td>51</td>
<td>Sink</td>
<td>mov eax, [edi+edx*4]</td>
</tr>
</tbody>
</table>

We use two approaches to generate a slice that captures the loop instructions for each optimization level. The first is a naive approach in which we calculate a backward slice between the functions input value, \( \text{arg}_0 \), and the final assignment to \( \text{eax} \), the return value. As explained in Section 4.4.4, we perform this analysis only on binaries compiled at optimization levels 1 and 2. Slicing between reading \( \text{arg}_0 \) and writing the final \( \text{eax} \) represents a slice from the input to the output. Ultimately, we have to unroll each loop in order to model the linear execution of the assembly language. This is the same limitation we discovered trying to S2E [Chipounov et al. 2011] or BAP [Brumley et al. 2011] for this research. We discuss this further in Section 4.5.4. These slices are listing alongside their semantics in Appendix C.

The second approach takes a slice from the \textit{source} and \textit{sink} inside the loop. Listings 4.18 and 4.19 show these slices from optimization levels 1 and 2 binaries.
Listing 4.15: Backward slice of fib() optimization level 0 from line 17 to line 35

1  mov   eax, [ebp+arg_0]
2  add   eax, 1
3  lea   edx, [eax-1]
4  mov   [ebp+var_10], edx
5  shl   eax, 2
6  lea   edx, [eax+3]
7  mov   eax, 10h
8  sub   eax, 1
9  add   eax, edx
10 mov   [ebp+var_1C], 10h
11 mov   edx, 0
12 div   [ebp+var_1C]
13 imul  eax, 10h
14 sub   esp, eax
15 mov   eax, esp
16 add   eax, 3
17 shr   eax, 2
18 shl   eax, 2
19 mov   [ebp+var_14], eax

Listing 4.16: Backward slice of fib() optimization level 1 from line 10 to line 17

1  mov   ebx, [ebp+arg_0]
2  lea   eax, ds:16h[ebx*4]
3  and   eax, 0FFFFFFFF0h
4  sub   esp, eax
5  mov   eax, esp
6  shr   eax, 2
7  lea   esi, ds:0[eax*4]
8  mov   dword ptr ds:0[eax*4], 0

Listing 4.17: Backward slice of fib() optimization level 2 from line 12 to line 21

1  mov   esi, [ebp+arg_0]
2  add   esi, 1
3  lea   eax, ds:12h[esi*4]
4  and   eax, 0FFFFFFFF0h
5  sub   esp, eax
6  mov   eax, esp
7  shr   eax, 2
8  mov   dword ptr ds:0[eax*4], 0
Listing 4.18: Backward slice of fib() optimization level 1 from line 24 to line 26

```
1     mov     ecx, [eax+4]
2     add     ecx, [eax]
3     mov     [eax+8], ecx
```

Listing 4.19: Backward slice of fib() optimization level 2 from line 32 to line 37

```
1     mov     ecx, [eax+4]
2     mov     ebx, [eax]
3     add     ecx, ebx
4     add     edx, 1
5     mov     [eax+8], ecx
```

4.4.3 Step 2: Extracting Semantics

After calculating slices using forward and backward slicing between sources and sinks, we translate each statement into an intermediate representation using Python. Most of the instructions contained in the slices described above are assignments and mathematical calculations, which simplifies the translation process.

Using symbolic bit vectors to represent unknown register and memory values also simplifies the modeling of certain instructions since the Z3 Python API overloads the corresponding Python functions with functions that mimic the machine instruction. Examples of this are the shifting instructions, \texttt{shl} and \texttt{slr}, where we are able to model \texttt{shl} using the overloaded \texttt{«} operator and \texttt{shr} using the overloaded \texttt{»} operator. The Z3 Python API also provides functions such as unsigned division, \texttt{UDiv()}, used on line 24 of Listing 4.20 for use with symbolic bit vectors.

Listing 4.20 contains slices from optimization levels 0 and 1 that perform the array size and address calculation and Listing 4.23 contains the same slices from optimization levels 1 and 2. Listings C.1, C.2, C.3, and C.4 show the slices obtained from unrolling the loop in optimization levels 1 and 2, and Listing 4.25 contains the slices that capture the internal
loop semantics from optimization levels 1 and 2. These listing also contain the Python code that executes the Z3 solver to test for equality between the pairs of slices.

4.4.4 Step 3: Testing Equality

The first slices we compare are the slices that calculate the size and address for the fib[] array. Lines 21-39 of Listing B.1 and lines 14-21 of Listing B.2 calculate the array size and address for the unoptimized binary and the binary compiled at optimization level 1. After slicing and extracting the semantics, we obtain the equations in listing 4.21 as the symbolic values of the array address at the end of each slice. The solver cannot prove these slices are equal and outputs a counterexample of values for arg_0 and esp.

Listing 4.20: Fibonacci array address calculation using optimization levels 0 and 1

```python
from z3 import *
s = Solver()
esp = BitVec('esp', 32)
# Since both functions modify esp, we create copies for each function.
esp_0 = esp
esp_1 = esp
arg_0 = BitVec('arg_0', 32)
# fib.c compiled with -m32 -O0
eax = arg_0 # mov eax, [ebp+arg_0]
eax = eax + 0x1 # add eax, 1
edx = eax - 0x1 # lea edx, [eax-1]
var_10 = edx # mov [ebp+var_10], edx
eax = eax << 0x2 # shl eax, 2
edx = eax + 0x3 # lea edx, [eax+3]
eax = 0x10 # mov eax, 10h
eax = eax - 0x1 # sub eax, 1
eax = eax + edx # add eax, edx
var_1C = 0x10 # mov [ebp+var_1C], 10h
edx = 0 # mov edx, 0
eax = UDiv(eax, var_1C) # div [ebp+var_1C]
eax = eax * 0x10 # imul eax, 10h
esp_0 = esp_0 - eax # sub esp, eax
eax = esp_0 # mov eax, esp
# This add instruction is the only semantic difference between the two optimization levels.
# Uncommenting this line prevents the solver from proving the equality of these two slices.
```
The reason these slices are not equal is that Line 32 of Listing 4.20, \texttt{add eax, 3}, causes the slice of instructions from the unoptimized binary to differ semantically from the binary compiled at optimization level 1. This was a calculation inserted by the compiler and, most likely, represents a rounding and byte alignment calculation. This is an example of a semantic area where the source code does not define behavior, but that the compiler is free to make a decision about memory locations and usage. This is a limitation we identified when comparing semantics extracted from data flow. Instructions resulting from a memory operation in a higher level source code language (where explicit memory control is not exposed to the developer) could contain semantic differences between non-optimized and optimized compilations. After removing the \texttt{add eax, 3} instruction from the unoptimized
slice, the solver is able to prove the equations are equal, shown in Listing 4.22. We cannot
definitely determine the significance of this addition instruction within the unoptimized slice,
but it illustrates that testing for similar functions using semantics can fail when the compiler
introduces differences not controlled by the source code.

Listing 4.22: Output of Listing 4.20 excluding Line 32

<table>
<thead>
<tr>
<th>Optimization Level 0:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(esp - UDiv(15 + (arg_0 + 1 &lt;&lt; 2) + 3, 16) * 16 &gt;&gt; 2) &lt;&lt; 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Optimization Level 1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(esp - (arg_0 * 4 + 22 &amp; 4294967280) &gt;&gt; 2) * 4 + 0</td>
</tr>
</tbody>
</table>

proved

To compare the array address calculation from optimization levels 1 and 2, we compare
the slices from Listings 4.16 and 4.17. Listing 4.23 shows the python script that compares
this slice with the same slice from the binary compiled at optimization level 1. Listing 4.24
contains the output from Listing 4.23 showing the “proved” result from the solver, meaning
these slice are equal.

Listing 4.23: Fibonacci array address calculation using optimization levels 1 and 2

<table>
<thead>
<tr>
<th>1</th>
<th>from z3 import *</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>s = Solver()</td>
</tr>
<tr>
<td>3</td>
<td>#fib.c compiled using -m32 -O1</td>
</tr>
<tr>
<td>4</td>
<td>#declare symbolic variables</td>
</tr>
<tr>
<td>5</td>
<td>arg_0 = BitVec(&quot;arg_0&quot;,32)</td>
</tr>
<tr>
<td>6</td>
<td>esp = BitVec(&quot;esp&quot;,32)</td>
</tr>
<tr>
<td>7</td>
<td>ebx = arg_0</td>
</tr>
<tr>
<td>8</td>
<td>eax = ebx * 4 + 0x16</td>
</tr>
<tr>
<td>9</td>
<td>eax = eax &amp; 0xffffffff0</td>
</tr>
<tr>
<td>10</td>
<td>esp = esp - eax</td>
</tr>
<tr>
<td>11</td>
<td>eax = esp - eax</td>
</tr>
<tr>
<td>12</td>
<td>esp = esp</td>
</tr>
<tr>
<td>13</td>
<td>eax = eax &gt;&gt; 0x2</td>
</tr>
<tr>
<td>14</td>
<td>fib1 = eax</td>
</tr>
<tr>
<td>15</td>
<td>#mov ebx, [ebp+arg_0]</td>
</tr>
<tr>
<td>16</td>
<td>#lea eax, ds:16h[ebx*4]</td>
</tr>
<tr>
<td>17</td>
<td>#and eax, 0xFFFFFFFF0h</td>
</tr>
<tr>
<td>18</td>
<td>#sub esp, eax</td>
</tr>
<tr>
<td>19</td>
<td>#mov eax, esp</td>
</tr>
<tr>
<td>20</td>
<td>#shr eax, 2</td>
</tr>
<tr>
<td>21</td>
<td>#lea esi, ds:0[eax*4]</td>
</tr>
<tr>
<td>22</td>
<td>#mov dword ptr ds:0[eax*4], 0</td>
</tr>
</tbody>
</table>

#fib.c compiled using -m32 -O2
Testing the looping behavior is more difficult than the array calculation. Creating slices according to our method that can be extracted to semantics and solved using an SMT solver is a difficult task because using a symbolic variable for a loop counter will cause the solver to run indefinitely. Our idea of using instructions that affect control flow to create constraints on the data flow was successful but required that we unroll the loop the same number of time in both functions before testing for equality. To our knowledge, no SMT solver supports looping in the way we would need to use it to model looping behavior using symbolic variables without unrolling loops to create non-cyclic models of the instructions.

For the naive slicing approach to capture the loop discussed in Section 4.4.2, We unroll the loop within the fib() function 0, 1, 2, and 3 times from the binaries compiled at
optimization levels 1 and 2. We use the information gained from control flow to determine constraints for the symbolic variables and unroll the loop equal numbers of times in both binaries, verifying the output after each unrolling. The Python scripts in Appendix C model slices of instructions starting at \texttt{arg_0} and ending at the return value of the \texttt{fib()} function. Each of the Python scripts in Appendix C illustrates that manually unrolling the loop in the binaries compiled at optimization levels 1 and 2 can prove the slices are equal.

A better solution that does not require unrolling loops is to calculate smaller slices and try to determine if subsets of the function’s behavior match. Using the slices from Listings 4.18 and 4.19 we are able to compare a data flow that performs the addition operation contain within the Fibonacci loop, reavealing that the behavior of these two slices is identical. Listing 4.25 shows a Python script testing the slices and semantics using the Z3 SMT solver. Listing 4.26 contains the output from this script.

Listing 4.25: Fibonacci loop semantics from optimization levels 1 and 2

```
from z3 import *

s = Solver()

# fib.c compiled using -m32 -O1
# declare symbolic variables
eax = BitVec("eax" ,32)
mem = Array("mem", BitVecSort(32), BitVecSort(32))
arg_0 = BitVec("arg_0",32)

ecx = mem[ eax +4]  # mov ecx, [eax+4]
ecx = mem[ eax]  + ecx  # add ecx, [eax]
mem = Store(mem, eax+8, ecx)  # mov [eax+8], ecx
fib1 = mem

# fib.c compiled using -m32 -O2
mem = Array("mem", BitVecSort(32), BitVecSort(32))

ecx = mem[ eax+4]  # mov ecx, [eax+4]
ebx = mem[ eax]  # mov ebx, [eax]
ecx = ecx + ebx  # add ecx, ebx
mem = Store(mem, eax+8, ecx)  # mov [eax+8], ecx
fib2 = mem
```
4.4.5 Conclusion

Use Case 2 includes a loop that is dependent on an input argument, which made calculating the exact behavior of the loop difficult since the input argument is represented as a symbolic variable. While we were able to prove the entire functions were semantically equal by unrolling the loop a static number of times, this also put unreasonable constraints on the input variable since the input variable decided the number of times the loop is unrolled. A better solution was to capture a smaller slice within the loop to compare the behavior inside the loop. The implications of this approach is that Data Flow Binary Matching is able to match some of the behavior of these functions, but not all of their behavior. This is an acceptable result for our research since a partial match is a better solution than no match at all, which is the result of the other binary matching techniques.

4.5 Use Case 3: Applying Data Flow Binary Differencing to Actual Malware

Use Case 3 applies our research to live malware. We chose this example to provide a real world look at how using Data Flow Binary Differencing might apply to a realistic problem faced by a reverse engineer. We compile the malware from source code at various
optimization levels so we can introduce syntactic differences into the malware’s functions without changing the semantics. Metamorphic malware uses these same techniques to evade signature-based anti-virus detection techniques without relying on encryption. [Walenstein et al. 2007]

The malware we are analyzing is sdbot version 0.5b. Sdbot is the client software installed on victim computer used to create a botnet, a large collection of infected computers an attacker can use for malicious actions such as launching a distributed denial of service (DDoS) attack. This malware is classified as a Trojan horse, a type of malware that tricks the user into installing itself usually through social engineering. Once installed and running on the victim computer, sdbot opens a backdoor that allows an attacker to perform various malicious actions on the infected system, such as stealing data or taking remote control of the victim computer. Sdbot is controlled through IRC (Internet Chat Relay), by connection to an IRC server and joining a chat room where the attacker is able to issue commands. IRC requires that each user connected to a server have a unique nickname so sdbot generates a random nickname before connecting to a IRC server. This nickname is the identity of each bot in the chat room and how the attacker can interact with each specific bot if desired. We chose the function that randomly generates this nickname as the target of our analysis. The source code for this function is shown in Listing 4.27.

The \texttt{rndnick()} function takes one argument, a pointer to a string buffer, returns a pointer to a string buffer, and contains 3 local variables, two integers and a 12 character length array. A number of system calls are used inside this functions. \texttt{GetTickCount()} is passed to \texttt{srand()} in order to initialize the random number generator. Next, \texttt{memset()} writes zeros to the 12 character long array, \texttt{nick[]}. A \texttt{for()} loop is used to fill the \texttt{nick[]} array using random ASCII characters between \texttt{nick[0]} and \texttt{nick[4]} to \texttt{nick[7]} depending on the value of \texttt{nl}, followed by a NULL string terminator. This results is a random string of lowercase ASCII character either between 4 and 7 character in length. Finally the 12 characters of \texttt{nick[]} are copied to \texttt{strbuf} and the pointer to \texttt{strbuf} is returned.
Listing 4.27: Source code for rndnick function

```c
char * rndnick (char * strbuf)
{
    int n, nl;
    char nick[12];

    srand(GetTickCount());
    memset(nick, 0, sizeof(nick));
    nl = (rand() % 3) + 4;
    for (n = 0; n < nl; n++) nick[n] = (rand() % 26) + 97;
    nick[n + 1] = '\0';

    strncpy(strbuf, nick, 12);
    return strbuf;
}
```

The full assembly language for `rndnick()` compiled at optimization level 0 is shown in Appendix B Listing B.4 and compiled at optimization level 1, in Listing B.5. Figures 4.4 and 4.5 show the Control Flow Graphs for these functions.

### 4.5.1 Analysis & Observations

Both `rndnick()` functions from optimization levels 1 and 2 have different control flow graphs so Structural Analysis Matching fails. Table 4.6 shows the structural metadata for `rndnick()`. Also both functions have different instructions for each of their basic blocks, therefore signatures generated for each basic block will be different and Signature Hash Map Matching also fails.

<table>
<thead>
<tr>
<th>Optimization Level</th>
<th>Number of Blocks</th>
<th>Edges</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>7</td>
<td>5</td>
</tr>
</tbody>
</table>

One syntactic and semantic difference we notice between the binaries is the systems call to `memset` has been removed from the binary compiled at optimization level 1. The compiler has decided since the `rndnick()` function does not use all 12 characters it allocates, and since
Figure 4.4: Control Flow Graph of rndnick() optimization level 0
Figure 4.5: Control Flow Graph of rndnick() optimization level 1
line 10 of source code, Listing 4.27, adds a string terminating NULL, zeroing out the array isn’t necessary. At optimization level 1, the compiler removes this system call replacing it with a couple of mov instructions writing zeros to the start of the nick[] array. The intent of the source code was to write zeros to all 12 memory addresses of the nick[] array. This is a semantic change internal to the function although is remedied by the call _strncpy at the end of the function. The function _strncpy will write zeros to the end of the destination if the source string is less than the number of bytes passed as an argument to _strncpy. In this case, _strncpy is called with 12 bytes as the num argument. The _memset functions actually performs redundant actions with _strncpy so the compiler has removed it.

4.5.2 Step 1: Slicing Instructions

We perform slicing on both binaries in the same way as the previous two use cases. We begin by identifying the sources and sinks within both binaries. Tables 4.7 and 4.8 show the sources and sinks from the unoptimized and level 1 optimized binaries, respectively. Line number are from Listings B.4 and B.5.

The call instructions are identified as sources in both functions. More specifically, the value of eax after the call is executed is the data we use when slicing the instructions following the call instruction and the arguments used by the call instructions are sinks. This allows for the capture of data flow to and from a sub-function.

Listings 4.28, 4.29, 4.30, 4.31, 4.32, 4.33, and 4.34 contain the slices obtained from the unoptimized binary and Listings 4.35, 4.36, 4.37, 4.38, and 4.39 show the slices from the level 1 optimized binary.

Listing 4.28, 4.29, 4.33, 4.35, and 4.38 all result from backward slicing from identified input argument to the listed call instruction. Backward slicing is the only slicing technique that works for capturing data flow to a called sub-function. We do not initially know how
Table 4.7: Use Case 3: Sources and Sinks for rndnick() optimization level 0

<table>
<thead>
<tr>
<th>Line number</th>
<th>Type</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>Source</td>
<td>call _GetTickCount</td>
</tr>
<tr>
<td>17</td>
<td>Source</td>
<td>call _srand</td>
</tr>
<tr>
<td>22</td>
<td>Source</td>
<td>call _memset</td>
</tr>
<tr>
<td>23</td>
<td>Source</td>
<td>call _rand</td>
</tr>
<tr>
<td>43</td>
<td>Source</td>
<td>call _rand</td>
</tr>
<tr>
<td>58</td>
<td>Source</td>
<td>mov edx, [ebp+var_C]</td>
</tr>
<tr>
<td>64</td>
<td>Source</td>
<td>mov eax, [ebp+var_C]</td>
</tr>
<tr>
<td>67</td>
<td>Source</td>
<td>mov eax, [ebp+var_C]</td>
</tr>
<tr>
<td>73</td>
<td>Source</td>
<td>mov eax, [ebp+arg_0]</td>
</tr>
<tr>
<td>75</td>
<td>Source</td>
<td>call _strncpy</td>
</tr>
<tr>
<td>76</td>
<td>Source</td>
<td>mov eax, [ebp+arg_0]</td>
</tr>
<tr>
<td>16</td>
<td>Sink</td>
<td>mov [esp], eax</td>
</tr>
<tr>
<td>18</td>
<td>Sink</td>
<td>mov dword ptr [esp+8], 0Ch</td>
</tr>
<tr>
<td>19</td>
<td>Sink</td>
<td>mov dword ptr [esp+4], 0</td>
</tr>
<tr>
<td>21</td>
<td>Sink</td>
<td>mov [esp], eax</td>
</tr>
<tr>
<td>37</td>
<td>Sink</td>
<td>mov [ebp+var_10], eax</td>
</tr>
<tr>
<td>38</td>
<td>Sink</td>
<td>mov [ebp+var_C], 0</td>
</tr>
<tr>
<td>60</td>
<td>Sink</td>
<td>mov [edx], al</td>
</tr>
<tr>
<td>69</td>
<td>Sink</td>
<td>mov [ebp+eax+var_1C], 0</td>
</tr>
<tr>
<td>70</td>
<td>Sink</td>
<td>mov dword ptr [esp+8], 0Ch</td>
</tr>
<tr>
<td>72</td>
<td>Sink</td>
<td>mov [esp+4], eax</td>
</tr>
<tr>
<td>74</td>
<td>Sink</td>
<td>mov [esp], eax</td>
</tr>
<tr>
<td>76</td>
<td>Sink</td>
<td>mov eax, [ebp+arg_0]</td>
</tr>
</tbody>
</table>
Table 4.8: Use Case 3: Sources and Sinks for rndnick() optimization level 1

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Type</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>Source</td>
<td>call _GetTickCount</td>
</tr>
<tr>
<td>22</td>
<td>Source</td>
<td>call _srand</td>
</tr>
<tr>
<td>26</td>
<td>Source</td>
<td>call _rand</td>
</tr>
<tr>
<td>44</td>
<td>Source</td>
<td>call _rand</td>
</tr>
<tr>
<td>70</td>
<td>Source</td>
<td>mov edx, [esp+3Ch+arg_0]</td>
</tr>
<tr>
<td>72</td>
<td>Source</td>
<td>call _strncpy</td>
</tr>
<tr>
<td>73</td>
<td>Source</td>
<td>mov eax, [esp+3Ch+arg_0]</td>
</tr>
<tr>
<td>21</td>
<td>Sink</td>
<td>mov [esp+3Ch+var_3C], eax</td>
</tr>
<tr>
<td>23</td>
<td>Sink</td>
<td>mov dword ptr [esp+3Ch+var_28], 0</td>
</tr>
<tr>
<td>24</td>
<td>Sink</td>
<td>mov [esp+3Ch+var_24], 0</td>
</tr>
<tr>
<td>25</td>
<td>Sink</td>
<td>mov [esp+3Ch+var_20], 0</td>
</tr>
<tr>
<td>54</td>
<td>Sink</td>
<td>mov [ebx], cl</td>
</tr>
<tr>
<td>66</td>
<td>Sink</td>
<td>mov [esp+ebp+3Ch+var_28+1], 0</td>
</tr>
<tr>
<td>67</td>
<td>Sink</td>
<td>mov [esp+3Ch+var_34], 0Ch</td>
</tr>
<tr>
<td>69</td>
<td>Sink</td>
<td>mov [esp+3Ch+var_38], eax</td>
</tr>
<tr>
<td>71</td>
<td>Sink</td>
<td>mov [esp+3Ch+var_3C], eax</td>
</tr>
<tr>
<td>73</td>
<td>Sink</td>
<td>mov eax, [esp+3Ch+arg_0]</td>
</tr>
</tbody>
</table>

far back the arguments are calculated so we must follow the identified input arguments backwards until no more use-define pairs exist in the slice.

Listing 4.30, 4.31, 4.32, 4.36, and 4.37 were generated using forward slicing between a source and sink. This creates a series of instructions where the definition of the data at the source and subsequent use-define pairs influence the data use at the sink.

Listing 4.34 and 4.39 are obtained by slicing between the function’s input arguments arg_0 and the function’s return values.

Listing 4.28: Optimization Level 0: Slice created from return value of _getTickCount@0 and argument of _srand

1 call _GetTickCount@0
2 mov [esp], eax ; unsigned int
3 call _srand
Listing 4.29: Optimization Level 0: Slice created from arguments to `_memset`

1. mov    dword ptr [esp+8], 0Ch ; size_t
2. mov    dword ptr [esp+4], 0 ; int
3. lea    eax, [ebp+var_1C]
4. mov    [esp], eax ; void *
5. call   `_memset`

Listing 4.30: Optimization Level 0: Slice created from return value of `_rand` at line 23

1. call   `_rand`
2. mov    ecx, eax
3. mov    eax, ecx
4. imul   edx
5. mov    eax, ecx
6. sar    eax, 1Fh
7. sub    edx, eax
8. mov    eax, edx
9. add    eax, eax
10. add   eax, edx
11. sub   ecx, eax
12. mov   edx, ecx
13. lea   eax, [edx+4]
14. mov   [ebp+var_10], eax

Listing 4.31: Optimization Level 0: Slice created from return value of `_rand` on line 43

1. call   `_rand`
2. mov    ecx, eax
3. mov    eax, ecx
4. imul   edx
5. sar    edx, 3
6. mov    eax, ecx
7. sar    eax, 1Fh
8. sub    edx, eax
9. mov    eax, edx
10. imul  eax, 1Ah
11. sub   ecx, eax
12. mov   eax, ecx
13. add   eax, 61h
14. mov   [edx], al
Listing 4.32: Optimization Level 0: Slice created from source at line 67

```
1  mov    eax, [ebp+var_C]
2  add    eax, 1
3  mov    [ebp+eax+var_1C], 0
```

Listing 4.33: Optimization Level 0: Slice created from the arguments to _strncpy at line 75

```
1  mov    dword ptr [esp+8], 0Ch ; size_t
2  lea    eax, [ebp+var_1C]
3  mov    [esp+4], eax ; char *
4  mov    eax, [ebp+arg_0]
5  mov    [esp], eax ; char *
6  call   _strncpy
```

Listing 4.34: Optimization Level 0: Slice created from arg_0 and return value.

```
1  mov    eax, [ebp+arg_0]
2  retn
```

Listing 4.35: Optimization Level 1: Slice created from return value of _getTickCount@0 and argument of _srand

```
1  call   _GetTickCount@0
2  mov    [esp+3Ch+var_3C], eax ; unsigned int
3  call   _srand
```

Listing 4.36: Optimization Level 1: Slice created from return value of _rand at line 26

```
1  call   _rand
2  mov    ecx, eax
3  imul   edx
4  mov    eax, ecx
5  sar    eax, 1Fh
6  sub     edx, eax
7  lea    eax, [edx+edx*2]
8  sub     ecx, eax
9  lea    eax, [ecx+4]
```
Listing 4.37: Optimization Level 1: Slice created from return value of _rand at line 44

```
1  call _rand
2  mov ecx, eax
3  imul esi
4  sar edx, 3
5  mov eax, ecx
6  sar eax, 1Fh
7  sub edx, eax
8  imul edx, 1Ah
9  sub ecx, edx
10 add ecx, 61h
11 mov [ebx], cl
```

Listing 4.38: Optimization Level 1: Slice created from arguments to _strncpy

```
1  mov [esp+3Ch+var_34], 0Ch ; size_t
2  lea eax, [esp+3Ch+var_28]
3  mov [esp+3Ch+var_38], eax ; char *
4  mov eax, [esp+3Ch+arg_0]
5  mov [esp+3Ch+var_3C], eax ; char *
6  call _strncpy
```

Listing 4.39: Optimization Level 1: Slice created from arg_0 and return value

```
1  mov eax, [esp+3Ch+arg_0]
2  retn
```

4.5.3 Step 2: Extracting Semantics

As with the previous use cases, we take each line of the slice and transform its behavior into Python code using symbolic variables where we do not know actual values. We compare and analyze each slice from Listings 4.28 to 4.39 in Section 4.5.4.

Translating the imul instruction to Python was cumbersome since the semantics of the instruction differ depending on the number of operands and the operands are symbolic. According to [Intel Corporation 2014], the imul instruction, when used with a single operand, takes the operand and multiplies it with the contents of the eax register placing the result
in the concatenation of the \texttt{edx} and \texttt{eax} registers. In mathematical terms, for \texttt{imul esi} this looks like \texttt{esi*eax = [edx:eax]} where \( : \) represents concatenation of the two registers. This was difficult to model explicitly in Python since we are using symbolic variables, but the Z3 solver’s Python API provides a few functions that allow us to implement the \texttt{imul} instruction accurately. We were able to zero extend the two factors so that the result product was actually 64-bits wide. Then we extracted the two 32-bit halves of the product into the \texttt{edx} and \texttt{eax} registers. Listing 4.40 shows the results of this implementation.

Listing 4.40: Modeling the x86 “imul esi” instruction in Python


definitions: \( \texttt{ZeroExt} \) and \( \texttt{Extract} \)

\begin{verbatim}
1  r = ZeroExt(64, esi) * ZeroExt(64, eax)
2  edx = Extract(63,32, r)
3  eax = Extract(31,0, r)
\end{verbatim}

4.5.4 Step 3: Testing for Equality

Comparing the slices from Listings 4.28 to 4.35, 4.33 to 4.38, and 4.34 to 4.39, we can see that these slices are identical, if certain variables are equal between the two optimization levels, assuming data exists at the same location in both functions.

Comparing Listings 4.28 to 4.35, if \texttt{esp} from 4.28 equals \texttt{esp+3Ch+var_3C} in 4.35, the slices are identical. The label \texttt{var_3C} is inserted by IDA and represents a constant value, \(-3Ch\). \texttt{3Ch+var_3C} means \texttt{3Ch-3Ch = 0} so these slices are equal since \texttt{esp} is the same as \texttt{esp+0}

Considering 4.33 and 4.38, if the equations in table 4.9 are valid, with values on the left from 4.33 and values on the right from 4.38, these slices are also equal.

We have already discussed that the \texttt{var_} labels followed by a hex value represents subtraction by that hex value, so we can the the first, third, and last equations in Table 4.9 are equal when we replace the \texttt{var_} with the negative hex value and solve.
Table 4.9: Equations that make Listings 4.33 and 4.38 equal.

\[
\begin{align*}
\text{esp}+8 & \quad \equiv \quad \text{esp}+3\text{Ch}+\text{var}_34 \\
\text{ebp}+\text{var}_1\text{C} & \quad \equiv \quad \text{esp}+3\text{Ch}+\text{var}_28 \\
\text{esp}+4 & \quad \equiv \quad \text{esp}+3\text{Ch}+\text{var}_38 \\
\text{ebp}+\text{arg}_0 & \quad \equiv \quad \text{esp}+3\text{Ch}+\text{arg}_0 \\
\text{esp} & \quad \equiv \quad \text{esp}+3\text{Ch}+\text{var}_3\text{C}
\end{align*}
\]

We can confirm the second and fourth equations are equal by looking at the instructions that modify \textit{esp} at the start of the \textit{rndnick()} function from the binary compiled at optimization level 0 shown in Listing 4.41. At the start of the \textit{rndnick()} function, the \textit{ebp} and \textit{esp} registers point to the base of the stack. Once data is pushed on the stack the \textit{esp} register will be decremented so that it points to the next free memory address on the stack. The \textit{push} instruction in Listing 4.41 subtracts 4 bytes from \textit{esp} and the \textit{sub exp, 38h} instruction subtracts \textit{38h} bytes from \textit{esp}. The difference between \textit{ebp} and \textit{esp} is now \textit{-3Ch} bytes. This means \textit{ebp} in the \textit{rndnick()} function from the unoptimized binary is equal to \textit{esp+3Ch} in \textit{rndnick()} from the level 1 optimized binary. This proves the second and fourth equations in Table 4.9 are equal. By solving each of the equations in Table 4.9 we prove the slices in Listings 4.33 and 4.38 are equal, meaning both \textit{rndnick()} functions pass the same arguments to \_strncpy.

The slices in Listings 4.34 and 4.39 are equal if \textit{ebp} in 4.34 equals \textit{esp+3Ch} in 4.39. We just proved this from the previous analysis so we know these slices are also equal.

Listing 4.41: Calculating \textit{esp} in unoptimized \textit{rndnick()}

\begin{verbatim}
1 push ebp
2 mov ebp, esp
3 sub esp, 38h
\end{verbatim}

Next we compare the slices in Listings 4.30 and 4.31 from the unoptimized binary to the slices in Listings 4.36 and 4.37 from the level 1 optimized binary. These slice represent the data flow resulting from the \texttt{call _rand} instructions. After translating these slices to Python, we were able to prove their equality using the same process described in the other
use cases. Listing 4.42 and 4.43 contain the full Python scripts these slices and Listing 4.44 and 4.45 show the output of these scripts.

Listing 4.42: Python script that compares first _rand call in the unoptimized rndnick() to optimization level 1 rndnick()

```python
from z3 import *

rand = BitVec('rand', 32)
unk = BitVec('unk', 32)
s.add(rand >= 0)

# sdbox05b -O0

eax = rand  # call _rand
edx = unk  # edx is unknown

r = ZeroExt(64, edx) * ZeroExt(64, eax)
edx = Extract(63, 32, r)
eax = Extract(31, 0, r)

eax = eax + edx  # sub edx, eax

eax = eax + edx  # add eax, edx

eax = ecx - eax  # sub ecx, eax

eax = ecx  # mov edx, ecx

eax = eax + edx  # sub edx, eax

var_10 = eax  # mov [ebp+var_10], eax

rndnick0 = var_10

print "Optimization level 0: "
print rndnick0

# sdbox05b -O1

eax = eax + 4  # lea eax, [edx+4]

var_10 = eax  # mov [ebp+var_10], eax

rndnick0 = var_10

print "Optimization level 1: "
print rndnick0
```

62
Listing 4.43: Python script that compares the second _rand call in the unoptimized rndnick() to optimization level 1 rndnick()

```python
from z3 import *
s = Solver()
var_1C = BitVec('var_1C', 32)
var_C = BitVec('var_C', 32)
rand = BitVec('rand', 32)
mem = Array("mem", BitVecSort(32), BitVecSort(8))
unk = BitVec("unk",32)
s.add(rand >= 0)
# sdbox05b -O0
eax = rand
# call rand
ecx = eax
# mov ecx, eax
edx = unk
# edx is undefined
# assume edx matches unknown input from other function
eax = ecx
# mov eax, ecx
# imul edx
r = ZeroExt(64,edx) * ZeroExt(64,eax)
edx = Extract(63,32,r)
eax = Extract(31,0,r)
edx = LShR(edx, 3)
# sar edx, 3
eax = ecx
# mov eax, ecx
edx = edx - eax
# sub edx, eax
eax = edx
# mov eax, edx
eax = eax * 0x1A
# imul eax, 1Ah
ecx = ecx - eax
# sub ecx, eax
eax = ecx
# mov eax, ecx
eax = eax + 0x61
# add eax, 61h
c ecx = var_1C
# lea ecx, [ebp+var_1C]
edx = var_C
# mov edx, [ebp+var_1C]
edx = edx + ecx
# add edx, ecx
# mov [edx], al
mem = Store(mem, edx, Extract(7,0,eax))
rndnick0 = Extract(7,0,eax)
print "Optimization level 0: "
print rndnick0
```
Listing 4.44: Output of Listing 4.42

Optimization level 0:
Extract(63, 32, ZeroExt(64, unk)*ZeroExt(64, rand)) -
LShR(rand, 31) +
Extract(63, 32, ZeroExt(64, unk)*ZeroExt(64, rand)) -
LShR(rand, 31) +
Extract(63, 32, ZeroExt(64, unk)*ZeroExt(64, rand)) -
LShR(rand, 31) +
4
Optimization level 1:
Extract(63, 32, ZeroExt(64, unk)*ZeroExt(64, rand)) -
LShR(rand, 31) +
(Extract(63, 32, ZeroExt(64, unk)*ZeroExt(64, rand)) -
LShR(rand, 31))*
2 +
4
proved
Listing 4.45: Output of Listing 4.43

Optimization level 0:
\[
\text{Extract}(7, 0, \text{rand} -\left(\text{LShR}\left(\text{Extract}(63, 32, \text{ZeroExt}(64, \text{unk})*\text{ZeroExt}(64, \text{rand})), 3\right) - \text{LShR}(\text{rand}, 31)\right)*26 + 97)
\]

Optimization level 1:
\[
\text{Extract}(7, 0, \text{rand} -\left(\text{LShR}\left(\text{Extract}(63, 32, \text{ZeroExt}(64, \text{unk})*\text{ZeroExt}(64, \text{rand})), 3\right) - \text{LShR}(\text{rand}, 31)\right)*26 + 97)
\]

proved

We have proven equality of all the slices created from the level 1 optimized binary, but we have two slices, Listing 4.29 and 4.32, remaining from the unoptimized binary that we could not match. This means the data flow identified by these two slices was either not captured by the slices we created in the optimized binary or a semantic change exists. For an automated implementation of our technique, we would alert the user of these non-matched slices as well as the sources and sinks that do not belong to a matched slice.

We manually analyze these slices to understand the reason they were not matched. The slice in Listing 4.32 represents writing a zero to the end of the nick[] array. This same behavior is represented at line 66 of rndnick() from the level 1 optimized binary. Employing the same approach we used to prove Listing 4.33 and 4.38 are equal, we test that the calculated addresses this slice and this instruction are equal. This means solving the equation \(\text{ebp + ebp + var}_C + 1 + \text{var}_1C == \text{esp + ebp + 3Ch + var}_28 +1\). Setting \(\text{ebp}\) as constant, the only remaining variable is \(\text{esp}\) from the optimization level 1 binary. By
analyzing the instructions at the top of `rudnick()` that manipulate `esp`, we determine that 
\[ \text{ebp} = \text{esp} - 3Ch, \]
which plugged into the equation above satisfies the equally statement, meaning the slice represents the same behavior as the instruction.

The remaining slice, Listing 4.29, contains the list of instructions that setup the call to `_memset` discussed in Section 4.5.1. This missing data flow does represent a semantic difference, but after further analysis we determine that the behavior of these instructions were actually redundant with the call `_strncpy` instructions which explains why the compiler removed this data flow. Ultimately, this change did not represent a significant difference but is a semantic difference that exists between the two binaries.

4.5.5 Conclusion

This use case illustrates that semantic differencing can be used on real world problems faced by real world security engineers and reverse engineers. We were able to show how Data Flow Binary Differencing handles the `call` instruction and creates slices to capture behavior surrounding the use of sub-procedures. Although this use case also includes looping, we were able to prove the compared functions contained the same data flows without having to unroll the loops since linear slices within the loops were captured by the use of `sources` and `sinks`. 
Our research goal was to provide another means of testing two functions to determine if they are similar, a common requirement in reverse engineering binary software. We compared our research to fundamental, yet very different, approaches used to compare functions, Symbol Name Matching, Signature Hash Matching, and Structural Analysis Matching. Specifically, our research goal was to be able to prove that two functions were actually similar after these matching techniques failed to show the function’s similarities. By focusing on data flow, we were able to use the semantics of the function to test if they perform the same actions.

5.1 Research Contributions

Our research fills a gap in the currently available techniques for binary differencing. Table 5.1 compares the three techniques discussed in Chapter 2 with our technique. Symbol Name Matching is useful for analysis that focuses on the symbol names, but does not have the capability to detect differences within Functions, Instructions, or Semantics. Signature Hash Map Matching, Structural Analysis Matching, and Data Flow Binary Differencing can detect differences present in the Functions, but Structural Analysis Matching fails for functional changes that do not alter the control flow graph. Only Signature Hash Map Matching and Data Flow Binary Differencing are able to detect differences within a basic block. Finally, our technique is the only available approach that allow for comparing semantics using data flow.
Using data flow to identify semantically identical behavior is the key contribution for our research. Our technique does not depend on the structure or the syntax of the instructions, only the effects of the structure and syntax on the data. We do not consider this to be “the” answer to binary differencing, yet it can be an additional tool to aid the reverse engineering of software. When used along side other binary matching technique, Data Flow Binary Differencing can be applied where other techniques fail, to gain a higher confidence that two functions contain similarities or differences. We also were able to discover semantic differences introduced by the compiler due to optimization. Both Use Cases 2 and 3 contained semantic differences between compiled binaries. While the primary use of Data Flow Binary Matching is to detect semantic similarities between binaries, we also show that this technique can detect semantic differences.

5.2 Limitations and Challenges

The largest challenge our research faced was figuring out how to deal with the loops in Use Case 2 (Fibonacci) and Use Case 3 (sdbot’s $rndnick()$ function). Our solutions for Use Case 2 were, first unroll the loops and compare each unrolling to each other starting at the input argument and ending with the return value of the function, and second to calculate a slice from within the loop and representing unknown values as symbolic variables. Both methods were able to prove the calculated slices were equal. By unrolling the loop, we were able to symbolically execute the entire data flow from input to output of the function. A better solution was to look inside the loops for data flow and compare the semantics of

---

1Using data flow.
the data flow inside each loop. This does not give us a complete view of the behavior of the functions, but is enough to determine that the two functions are similar, achieving our research goal.

The loop contained in Use Case 3 did not create the same level of challenge as the loop in Use Case 2. The primary reason for this difference is the loop index in Use Case 2 was derived from the input argument for the function and had to be represented using a symbolic variable in our analysis. Had we implement the loop in Use Case 2 as it was derived directly from the semantics of the assembly language and if the SMT solver was able to loop over these statements, the solver would never finish since the loop terminating condition is symbolic. The same problem did not occur in Use Case 3 because the semantics within the loop of Use Case 3 could be contained within a single data flow. The loop index and the loop terminating condition are not part of the data flow inside the loop statements.

Another challenge we faced and were not able to overcome was comparing the calculation of the memory array in Use Case 2 between the optimization level 0 and 1 binaries. Our conclusion is that the two calculations are actually different. The compiler is freely able to calculate memory locations and addresses as long as the end result follows the intent of the source code. In Use Case 2, the unoptimized binary uses a slightly different method to calculate the starting memory address for the local array than the optimized binaries. Although this is a limitation of the Data Flow Binary Differencing technique, this is still a semantic difference discovered by our technique.

5.3 Future Work

The next step of our research is to automate the process of creating slices, transforming slices to an intermediate language, and then testing the slices for equality. The most likely path for this continuation of our research would be to incorporate Data Flow Binary Differencing into a tool like S2E [Chipounov et al. 2011] and BAP [Brumley et al. 2011], but these
tools would need to be modified to add the ability for to analyze multiple binaries at one time or be able to extract semantics and reuse them in the analysis of a different binary.

We believe that using semantic for reverse engineering is a very important research topic for the future. Our look at using semantics for binary differencing is just one area where applying semantics to reverse engineering can be used. Our work could be expanded to look at creating signatures based on semantics to be able to detect the presence of similar code in many binaries and allow a reverse engineer to detect code reuse or the theft of proprietary algorithms.

An area we did not include in this research was inter-procedural slicing. Future research should include research on how to implement inter-procedural slicing for Data Flow Binary Differencing so that we could also detect function in-lining. Currently we can only detect this as a semantic difference, but for a completely in-lined function, we should be able to use semantics to determine if the in-lining actually creates any differences. In order to do this we need to be able to begin a slice prior to a call instruction, follow the execution through the sub-procedure, and, using the return value of the sub-procedure, continue slicing until our sink is reached.

Beyond expanding the scope of this research, we also identified a few more narrow opportunities for further research. As we discuss in Section 5.1, an interesting contribution of this research was the discovery of semantical differences introduced by the compiler. The potential exists for compiler optimizations to introduce security vulnerabilities detectable as semantic differences. This research area could help discover very hard to detect vulnerabilities that result from optimizations, as well as classifying “safe” and “unsafe” compiler optimization and under what circumstances certain optimization could introduce vulnerabilities.

We could also further investigate semantic differences, such as the array address calculation in Use Case 2, that do not affect the program state. Being able to identify these types of differences could allow for a more precise implementation of Data Flow Binary Differencing
that is able to ignore these slight differences or at least report their existence to the reverse engineer. Knows that a semantic difference exists but that it doesn’t not affect the program state, such as placing a data structure at a byte higher address than a non-optimized implementation, allows the reverse engineer to focus on other important differences or at least judge the impact of this semantic change in the context of the rest of the program state.

Beyond Data Flow Binary Differencing, we also identified a few areas related to symbolic execution that warrant further research. Currently, an SMT solver designed for reverse engineering does not exist. There are many assembly language constructs that are difficult to translate into the solver specific language such as loops and jumps. Having a more natural means of expressing these constructs in a language an SMT solver can reason about would be a good addition to the currently available tools for program analysis using symbolic execution.
Bibliography


Appendices
Appendix A

Source Code

Listing A.1: Hello World source code

```c
#include <stdio.h>

int getNext(int num);

int main(){
    int i=0;
    while (i < 3){
        i = getNext(i);
        printf("%s", "Hello World!\n");
    }
    return 0;
}

int getNext(int num){
    return num++;
}
```

Listing A.2: Source code for fib.c

```c
#include <stdio.h>

int fib(int n)
{
    /* Declare an array to store fibonacci numbers. */
    int f[n+1];
    int i;
    /* 0th and 1st number of the series are 0 and 1*/
    f[0] = 0;
    f[1] = 1;
    for (i = 2; i <= n; i++)
    {
        /* Add the previous 2 numbers in the series and store it */
        f[i] = f[i-1] + f[i-2];
    }
```
```c
    return f[n];
  }

int main ()
{
  int n = 12;
  printf("%d", fib(n));
  return 0;
}
```
Listing B.1: Assembly of fib.c compiled with -O0

; SUBROUTINE
; Attributes: bp-based frame
public fib
fib proc near
var_1C = dword ptr -1Ch
var_14 = dword ptr -14h
var_10 = dword ptr -10h
var_C = dword ptr -0Ch
var_4 = dword ptr -4
arg_0 = dword ptr 8
push ebp
mov ebp, esp
push ebx
sub esp, 24h
mov eax, esp
mov ecx, eax
mov eax, [ebp+arg_0]
add eax, 1
lea edx, [eax-1]
mov [ebp+var_10], edx
shr eax, 2
lea edx, [eax+3]
mov eax, 10h
sub eax, edx
add eax, edx
mov [ebp+var_1C], 10h
mov edx, 0
div [ebp+var_1C]
imul eax, 10h
sub esp, eax
mov eax, esp
add eax, 3
shr eax, 2
shr eax, 2
mov [ebp+var_14], eax
mov eax, [ebp+var_14]
mov dword ptr [eax], 0
mov eax, [ebp+var_14]
mov dword ptr [eax+4], 1
mov [ebp+var_C], 2
jmp short loc_80484A9

loc_8048481:
Listing B.2: Assembly of fib.c compiled with -O1

```assembly
loc_80484A9:
    mov    eax, [ebp+var_C]
    cmp    eax, [ebp+arg_0]
    jle    short loc_8048481
    mov    eax, [ebp+var_14]
    mov    edx, [ebp+arg_0]
    mov    edx, [eax+edx*4]
    mov    ebx, edx
    mov    ecx, esp
    mov    eax, edx
    add    [ebp+var_C], 1
    loc_804845E:
    mov    ecx, [eax+edx*4]
    add    ecx, [eax]
    mov    eax, ecx
    mov    edx, 2
    lea    esi, ds:0[eax*4]
    mov    edx, ds:0[eax*4]
    cmp    ebx, 1
    jle    short loc_8048470
    mov    eax, esi
    mov    edx, 2
    add    ecx, [eax+edx*4]
    jmp    short loc_804845E:
    mov    ecx, [eax+edx*4]
    add    ecx, [eax]
```
Listing B.3: Assembly of fib.c compiled with -O2

```assembly
; Attributes: bp-based frame
public fib
fib proc near
arg_0 = dword ptr 8
push ebp
mov ebp , esp
push edi
push esi
push ebx
sub esp , 0Ch
mov esi , [ ebp + arg_0 ]
add esi , 1
lea eax , ds:12h[esi*4]
and eax , 0FFFFFFFF0h
sub esp , eax
mov eax , esp
shr eax , 2
cmp [ebp+arg_0], 1
lea edi , ds:0[eax*4]
mov dword ptr ds:0[eax*4], 0
mov dword ptr ds:4[eax*4], 1
jle short loc_80484AC
mov eax , edi
xor ebx , ebx
mov ecx , 1
mov edx , 2
jmp short loc_804849D
align 8
loc_8048498:
mov ecx , [eax+4]
mov ebx , [eax]
loc_804849D:
add ecx , ebx
add edx , 1
mov [eax+8], ecx
fib endp
```
Listing B.4: Assembly of sdbot05b.cpp compiled with -O0

```assembly
add   eax, 4
cmp   esi, edx
jnz   short loc_8048498
loc_80484AC:
mov   edx, [ebp+arg_0]
mov   eax, [edi+edx*4]
lea   esp, [ebp-0Ch]
pop   ebx
pop   esi
pop   edi
pop   ebp
retn
fib   endp
```

```assembly
; Attributes: bp-based frame
; _DWORD __cdecl rndnick(char *)
public __Z7rndnickPc
__Z7rndnickPc proc near

var_1C = byte ptr -1Ch
var_10 = dword ptr -10h
var_C = dword ptr -0Ch
arg_0 = dword ptr 8
push   ebp
mov    ebp, esp
sub    esp, 38h
call   _GetTickCount@0 ; GetTickCount()
mov    [esp], eax ; unsigned int
call   _srand
mov    dword ptr [esp+8], 0Ch ; size_t
mov    dword ptr [esp+4], 0 ; int
lea    eax, [ebp+var_1C]
mov    [esp], eax ; void *
call   _memset
call   _rand
mov    ecx, eax
mov    edx, 55555556h
mov    eax, ecx
imul   edx
mov    eax, ecx
sar    eax, 1Fh
sub    edx, eax
mov    eax, edx
add    eax, eax
add    eax, edx
sub    ecx, eax
mov    edx, ecx
lea    eax, [edx+4]
mov    [ebp+var_10], eax
```
Listing B.5: Assembly of sdbot05b.cpp compiled with -O1

1;=============== SUBROUTINE ==================
2;_DWORD __cdecl rndnick(char *)
3public __Z7rndnickPc
4__Z7rndnickPc proc near ;
var_3C = dword ptr -3Ch
var_38 = dword ptr -38h
var_34 = dword ptr -34h
var_28 = byte ptr -28h
var_24 = dword ptr -24h
var_20 = dword ptr -20h
arg_0 = dword ptr 4
push ebp
push edi
push esi
push ebx
sub esp , 2Ch
call _GetTickCount@0 ; GetTickCount()
mov [esp +3Ch+ var_3C], eax ; unsigned int
call _srand
mov dword ptr [esp+3Ch+var_28], 0
mov [esp+3Ch+var_24], 0
mov [esp+3Ch+var_20], 0
call _rand
mov ecx, eax
mov edx, 55555556h
imul edx
mov eax, ecx
sar eax, 1Fh
sub edx, eax
lea eax, [edx+edx*2]
sub ecx, eax
lea eax, [ecx+4]
test eax, eax
jle short loc_401D50
lea ebx, [esp+3Ch+var_28]
mov ebp, ecx
lea edi, [esp+ecx+3Ch+var_24]
mov esi, 4EC4EC4Fh
loc_401D27:
call _rand
mov ecx, eax
imul esi
sar edx, 3
mov eax, ecx
sar eax, 1Fh
sub edx, eax
imul edx, 1Ah
sub ecx, edx
add ecx, 61h
mov [ebx], cl
add ebx, 1
cmp ebx, edi
jnz short loc_401D27
add ebp, 4
jmp short loc_401D55
loc_401D50:


```
59       mov    ebp, 0
60 loc_401D55:
61       mov    [esp+ebp+3Ch+var_28+1], 0
62       mov    [esp+3Ch+var_34], 0Ch ; size_t
63       lea    eax, [esp+3Ch+var_28] ; 
64       mov    [esp+3Ch+var_38], eax ; char *
65       mov    eax, [esp+3Ch+arg_0]
66       mov    [esp+3Ch+var_3C], eax ; char *
67       call   _strncpy
68       mov    eax, [esp+3Ch+arg_0]
69       add    esp, 2Ch
70       pop     ebx
71       pop     esi
72       pop     edi
73       pop     ebp
74       retn
75 __Z7rndnickPc endp
```
Appendix C
Python scripts for unrolling the Fibonacci loop

Listing C.1: Python script that compares fib() -O1 and -O2 by unrolling 0 times

```python
from z3 import *
s = Solver()

# fib.c compiled using -O1
# declare variables
mem = Array("mem", BitVecSort(32), BitVecSort(32))
arg_0 = BitVec("arg_0",32)
esp = BitVec("esp",32)

# assume arg_0 >= 0 and arg_0 <= 1
s.add(arg_0 >= 0)
s.add(arg_0 <= 1)

ebx = arg_0
# mov ebx, [ebp+arg_0]
eax = (ebx * 4) + 0x16
# lea eax, ds:16h[ebx*4]
eax = eax & 0xFFFFF0
# and eax, 0xFFFFF0h
esp = esp - eax
# sub esp, eax
eax = esp
# mov eax, esp
eax = eax >> 2
# shr eax, 2
esi = eax * 4
# lea esi, ds:0[eax*4]
mem = Store(mem, (eax*4)+0, 0)
mem = Store(mem, (eax*4)+4, 1)

# cmp ebx, 1
# jle short loc_804870

# assum ebx <= 1
#------------------
eax = mem[esi+ebx*4]
# mov eax, [esi+ebx*4]
fib1 = eax

# fib.c compiled using -O2
# declare variables
mem = Array("mem", BitVecSort(32), BitVecSort(32))
arg_0 = BitVec("arg_0",32)
esp = BitVec("esp",32)

esi = arg_0
# mov esi, [ebp+arg_0]
esi = esi + 1
# add esi, 1
eax = (esi * 4) + 0x12
# lea eax, ds:12h[esi*4]
eax = eax & 0xFFFFF0
# and eax, 0xFFFFF0h
```
Listing C.2: Python script that compares fib() -O1 and -O2 by unrolling 1 time

```python
from z3 import *
s = Solver()

# fib.c compiled using -m32 -O1
# declare variables
mem = Array("mem", BitVecSort(32), BitVecSort(32))
arg_0 = BitVec("arg_0",32)
esp = BitVec("esp",32)

# assume arg_0 > 1 and arg_0 <= 2
s.add(arg_0 == 2)

ebx = arg_0
eax = (ebx * 4) + 0x16
eax = eax & 0xFFFFFFFF
esp = esp - eax
eax = esp >> 2
esi = eax * 4
mem = Store(mem, (eax*4)+0, 0)
mem = Store(mem, (eax*4)+4, 1)

# assum ebx <= 1
edx = esi
edx = 2
c ecx = mem[eax+4]
```

86
Listing C.3: Python script that compares fib() -O1 and -O2 by unrolling 2 times
from z3 import *

s = Solver()

# fib.c compiled using -m32 -O1
# declare variables
mem = Array("mem", BitVecSort(32), BitVecSort(32))
arg_0 = BitVec("arg_0",32)
esp = BitVec("esp",32)
s.add(arg_0 == 3)

eax = (ebx * 4) + 0x16  # lea eax, ds: 16h[ebx*4]
eax = eax & 0xFFFFFFFF  # and eax, 0xFFFFFFFF
esp = esp - eax  # sub esp, eax
esp = esp  # mov eax, esp
eax = eax >> 2  # shr eax, 2
esi = eax * 4  # lea esi, ds:0[eax*4]
mem = Store(mem, (eax *4) +0 , 0)  # mov dword ptr ds:0[eax*4]
mem = Store(mem, (eax *4) +4 , 1)  # mov dword ptr ds:0[eax*4]

# assume ebx <= 1
#-------------------
eax = esi  # mov eax, esi
edx = 2  # mov edx, 2
#-------------------
ecx = mem[eax+4]  # mov ecx, [eax+4]
mem = Store(mem, eax+8, ecx)  # mov [eax+8], ecx
edx = edx + 1  # add edx, 1
eax = eax + 4  # add eax, 4
# cmp ebx, edx
# jle short loc_804870
# assume ebx > edx (ebx == 3)
#-------------------
ecx = mem[eax+4]  # mov ecx, [eax+4]
mem = Store(mem, eax+8, ecx)  # mov [eax+8], ecx
edx = edx + 1  # add edx, 1
eax = eax + 4  # add eax, 4
# cmp ebx, edx
# jle short loc_804870

# ebx <= edx (ebx == 3)
#-------------------
eax = mem[esi+ebx*4]  # mov eax, [esi+ebx*4]

fib1 = eax

# fib.c compiled using -m32 -O2
Listing C.4: Python script that compares fib() -O1 and -O2 by unrolling 3 times

```python
from z3 import *
```

```python
prove (fib1 == fib2)
```
s = Solver()

#fib.c compiled using -m32 -O1
#declare variables
mem = Array("mem", BitVecSort(32), BitVecSort(32))
arg_0 = BitVec("arg_0",32)
est = BitVec("esp",32)
s.add(arg_0 == 4)

ebx = arg_0  # mov ebx, [ebp+arg_0]
eax = (ebx * 4) + 0x16  # lea eax, ds: 16h[ebx*4]
eax = eax & 0xFFFFFFF0  # and eax, 0FFFFFFFFh
esp = esp - eax  # sub esp, eax
eax = esp  # mov eax, esp
eax = eax >> 2  # shr eax, 2
esi = eax * 4  # lea esi, ds:0[eax*4]
mem = Store(mem, (eax*4)+0, 0)  # mov dword ptr ds:0[eax*4]
mem = Store(mem, (eax*4)+4, 1)  # mov dword ptr ds:0[eax*4]

# cmp ebx, 1
# jle short loc_804870
# assume ebx <= 1
#-----------------
eax = esi  # mov eax, esi
edx = 2  # mov edx, 2
#-----------------
ecx = mem[eax+4]  # mov ecx, [eax+4]
ecx = ecx + mem[eax]  # add ecx, [eax]
mem = Store(mem, (eax+8), ecx)  # mov [eax+8], ecx
edx = edx + 1  # add edx, 1
eax = eax + 4  # add eax, 4

# cmp ebx, edx
# jle short loc_804870
# assume ebx > edx (ebx == 4)
#-----------------
ecx = mem[eax+4]  # mov ecx, [eax+4]
ecx = ecx + mem[eax]  # add ecx, [eax]
mem = Store(mem, (eax+8), ecx)  # mov [eax+8], ecx
edx = edx + 1  # add edx, 1
eax = eax + 4  # add eax, 4

# cmp ebx, edx
# jle short loc_804870
# assume ebx > edx (ebx == 4)
#-----------------
ecx = mem[eax+4]  # mov ecx, [eax+4]
ecx = ecx + mem[eax]  # add ecx, [eax]
mem = Store(mem, (eax+8), ecx)  # mov [eax+8], ecx
edx = edx + 1  # add edx, 1
eax = eax + 4  # add eax, 4

# cmp ebx, edx
# jle short loc_804870
# assume ebx <= edx (ebx == 3)
# --------------
57  eax = mem[esi+ebx*4]  # mov eax, [esi+ebx*4]
59
60  fib1 = eax
62 #fib.c compiled using -m32 -O2
63 #declare variables
64  mem = Array("mem", BitVecSort(32), BitVecSort(32))
65  esp = BitVec("esp",32)
66
67  esi = arg_0  # mov esi, [ebp+arg_0]
68  esi = esi + 1  # add esi, 1
69  eax = (esi * 4) + 0x12  # lea eax, ds:12h[esi*4]
70  eax = eax & 0xFFFFFFFFF0h  # and eax, 0FFFFFFFF0h
71  esp = esp - eax  # sub esp, eax
72  eax = esp  # mov eax, esp
73  eax = eax >> 2  # shr eax, 2
74  edi = eax * 4 + 0  # lea edi, ds: 0[eax*4]
75  mem = Store(mem, (eax*4)+0, 0)  # mov dword ptr ds:0[eax*4], 0
76  mem = Store(mem, (eax*4)+4, 1)  # mov dword ptr ds:0[eax*4], 1
79  # assume arg_0 <= 1
81 # --------------
81  eax = edi  # mov eax, edi
82  ebx = 0  # xor ebx, ebx
83  ecx = 1  # mov ecx, 1
84  edx = 2  # mov edx, 2
86 # --------------
86  ecx = ecx + ebx  # add ecx, ebx
87  edx = edx + 1  # add edx, 1
88  mem = Store(mem,eax+8, ecx)  # mov [eax+8], ecx
89  eax = eax + 4  # add eax, 4
90  # cmp esi, edx
91  # assume esi != edx (esi == 4)
93 # --------------
93  ecx = mem[eax+4]  # mov ecx, [eax+4]
94  ebx = mem[eax]  # mov ebx, [eax]
96 # --------------
96  ecx = ecx + ebx  # add ecx, ebx
97  edx = edx + 1  # add edx, 1
98  mem = Store(mem,eax+8, ecx)  # mov [eax+8], ecx
99  eax = eax + 4  # add eax, 4
100  # cmp esi, edx
101  # assume esi != edx (esi == 4)
103 # --------------
103  ecx = mem[eax+4]  # mov ecx, [eax+4]
104  ebx = mem[eax]  # mov ebx, [eax]
106 # --------------
106  ecx = ecx + ebx  # add ecx, ebx
107  edx = edx + 1  # add edx, 1
108  mem = Store(mem,eax+8, ecx)  # mov [eax+8], ecx
109  eax = eax + 4  # add eax, 4
110  # cmp esi, edx
# assume esi == edx (esi == 4)
# --------------
edx = arg_0  # mov edx, [epb+arg_o]
eax = mem[edi+edx*4]  # mov eax, [edi+edx*4]
fib2 = eax
# Test for equality
prove (fib1 == fib2)