LEVERAGING PROGRAM SLICING TO UNDERSTAND NETWORK TRAFFIC

A Dissertation
Presented to
The Academic Faculty

by

Allen Joel Stewart

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
December 2019

LEVERAGING PROGRAM SLICING TO UNDERSTAND NETWORK TRAFFIC

Approved by:

Dr. Saltaformaggio, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Beyah
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Smith
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: [December 03, 2019]

ACKNOWLEDGEMENTS

I would like to express my deepest thanks to both the CyFI lab and GTRI-CIPHER for all of the support I received in pursuit of my Master's Degree and Thesis. Without the leadership and encouragement made available to me, this journey would not have been the same.

Dr. Saltaformaggio, thank you for affording me the opportunity to work for you in the CyFI lab while striving for my degree. You have done an amazing job in building a group of students that are both brilliant and eager to help. I appreciate the opportunity to learn from you and your team. You have pushed me to achieve things I did not believe I was capable of. I am forever grateful for this.

Mr. Roberts, thank you for the opportunity to work for you at GTRI while pursuing my graduate degree. It was there that found my passion for cyber security, and the trajectory of my career forever altered. I appreciate the guidance you have offered me, both academically and professionally. I look forward to the opportunity to continue learning and contributing to the work you are doing.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

angr	**The angr Python binary analysis framework**
VM	**Virtual Machine**
CFG	**Control Flow Graph**
CDG	**Control Dependency Graph**
DDG	**Data Dependency Graph**
PDG	**PROGRAM DEPENDENCY GRAPH**
BS	**Backward Slice**
repo	**A GitHub repository**
CyFI	**The Cyber Forensics Innovation Laboratory**
GTRI	**The Georgia Tech Research Institute**
CIPHER	**GTRI Cybersecurity, Information Protection, and Hardware Evaluation Research**
NVD	**The Network Vulnerability Division in GTRI-CIPHER**
CLI	**Command Line Interface**

SUMMARY

The goal of this project is to demonstrate how program slicing, specifically backward slice analysis, of binaries can be utilized to characterize the behavior of malicious software. This is accomplished, in part, by developing a modular toolkit that takes a binary as input and performs a backward slice on specific statements of interest. Specifically, these tools enable an analyst to review the contents of a function call. These tools are both modular and adaptable to flex into as many roles and functions as possible. This ethos is central to the utility of this project. Every component stands on its own to allow analysts to perform targeted analyses for specific cases. The tools are designed such that they do not constrain an analyst, but rather ease the reverse engineering burden upon them. When fully integrated, the tools developed for this project enable an analyst to focus their reverse engineering efforts more efficiently by clearly articulating the areas of the binary that are of the greatest interest.

A major motivation for the tools that are developed for this project is to be able to reconstruct the contents of a package that is sent by a malicious piece of software. This is useful for the following two reasons: identifying what information a bad actor has stolen from a system, and characterizing the type of data a bad actor expects to see. The latter is of value when spoofing malicious traffic. This project argues that analysis by way of program slicing, specifically backward slicing, is specifically suited for these tasks. This is accomplished by thoroughly explaining the concepts surrounding backward slice techniques, developing tools to perform analysis using backward slice analysis, demonstrate correctness and utility of such tools, and integrate parts of the tools into external analysis systems for other analysts to utilize.

CHAPTER 1. INTRODUCTION AND BACKGROUND

This project centers around utilizing the concept of backward slice analysis of a binary to reconstruct the contents of a function call towards function signature, arguments, and return type identification. To that end, a set of tools is developed to enable an analyst to glean relevant information about what data a binary is accessing and what the binary is doing with the data it accesses. Each tool plays a small role in this overarching goal while also being capable of standing on its own. Robust backward slice analysis functionality has utility in a variety of applications including detection of unauthorized gathering of personal information, malicious use of private functions, and unknown network protocol analysis. These use cases will be discussed in the paper.

This project aims to enable analysts to better characterize the behavior of malicious software through the lenses of backward slice analysis. The tools developed toward this end will explore and expand upon the current research employing backward slices. A functional integration of the tools developed and discussed in this project into a malware analysis workflow enable an analyst to more readily utilize backward slice analysis to better focus their efforts by pointing out the addresses of interest in a binary. This section details the information that can be gleaned from backward slice analysis and demonstrates that automated generation of such data can greatly reduce the burden of hand calculated analysis.

1.1 Backward Slice

Backward slice analysis, being central to the ideas presented herein, this section serves as an introduction to the concepts surrounding the backward slice. Beginning first with a detailed explanation of what a backward slice is. Followed by real world applications where backward slice analysis aids in reverse engineering efforts. Ending with a simple example that demonstrates how a backward slice is computer by hand. A solid grasp of these ideas is necessary to understand the work discussed throughout this paper.

1.1.1 What is a Backward Slice

A backward slice is a type of program slice that is used as a means by which to determine the statements involved in the computation of a variable at a particular address. It is a "backwards traversal of the program dependency graph" [1]. The program dependency graph is a combination of both the data dependency graph and control dependency graph. See Figure 4 – Simple Example Control and Data Dependency Graph for an example of a PDG.

1.1.2 When are Backward Slices Useful

Backward slice analysis is beneficial in applications where an analyst is interested in the origin of the definition of some variable in a program. Backward slices allow an analyst to see what data and instructions are influential in the state of a program at a given target. A target is the program address where backward slice is to begin computation. This type of information can be useful in reconstructing the contents of a network call, the data being written to a file, interactions between a peripheral and a device driver, and many other normal computing functions that could be exploited by malicious programs. By reconstructing this data in an easy-to-digest format, such as in a backward slice, the tools

developed for this project will enable analysts to more easily and quickly investigate for misbehaviour in these kinds of applications.

1.1.3 Example

This section shows an example of a backward slice using the simple example C source code shown in Figure 1 – Simple Example Source Code. It demonstrates how a backward slice is computed and what information the slice conveys. As a note, the code is compiled with GCC and run on the Intel based Ubuntu machine described in APPENDIX A. Development environment setup.

```
int main()
{
    char buf[1];
    char true = 'T';
    read(0, buf, 1);
    if (buf[0] == 'x')
    {
        buf[0] = true;
    }
    write(1, buf, 1);
}
```

Figure 1 - Simple Example Source Code.

Before beginning any analysis, it is useful to take a look at the object dump of the compiled code which can be seen below in Figure 2. This figure shows the instructions associated with the source code and allows for a more detailed look at what instructions are actually being executed.

```
6fa:   55                      push   %rbp
6fb:   48 89 e5                mov    %rsp,%rbp
6fe:   48 83 ec 20             sub    $0x20,%rsp
702:   89 7d ec                mov    %edi,-0x14(%rbp)
705:   48 89 75 e0             mov    %rsi,-0x20(%rbp)
709:   64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
710:   00 00
712:   48 89 45 f8             mov    %rax,-0x8(%rbp)
716:   31 c0                   xor    %eax,%eax
718:   c6 45 f6 54             movb   $0x54,-0xa(%rbp)            ; [char true = 'T';]
71c:   48 8d 45 f7             lea    -0x9(%rbp),%rax
720:   ba 01 00 00 00          mov    $0x1,%edx                   ; size of buffer = 1
725:   48 89 c6                mov    %rax,%rsi                   ; point to the buffer
728:   bf 00 00 00 00          mov    $0x0,%edi                   ; file descriptor = 0
72d:   e8 9e fe ff ff          callq  5d0 <read@plt>              ; [read (0, buf, 1);]
732:   0f b6 45 f7             movzbl -0x9(%rbp),%eax
736:   3c 78                   cmp    $0x78,%al                   ; [if (buf[0] == 'x');]
738:   75 07                   jne    741 <main+0x47>             ; not 'x', then 741, else 73a
73a:   0f b6 45 f6             movzbl -0xa(%rbp),%eax             ; is 'x', so retrieve 'T'
73e:   88 45 f7                mov    %al,-0x9(%rbp)              ; [buf[0] = true;]
741:   48 8d 45 f7             lea    -0x9(%rbp),%rax
745:   ba 01 00 00 00          mov    $0x1,%edx                   ; size of buffer = 1
74a:   48 89 c6                mov    %rax,%rsi                   ; pointer to the buffer
74d:   bf 01 00 00 00          mov    $0x1,%edi                   ; file descriptor = 1
752:   e8 59 fe ff ff          callq  5b0 <write@plt>             ; [write(1, buf, 1);]
757:   b8 00 00 00 00          mov    $0x0,%eax
75c:   48 8b 4d f8             mov    -0x8(%rbp),%rcx
760:   64 48 33 0c 25 28 00    xor    %fs:0x28,%rcx
767:   00 00
769:   74 05                   je     770 <main+0x76>
76b:   e8 50 fe ff ff          callq  5c0 <__stack_chk_fail@plt>
770:   c9                      leaveq
771:   c3                      retq
772:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
779:   00 00 00
77c:   0f 1f 40 00             nopl   0x0(%rax)
```

Figure 2 - Simple Example Object Dump.

The first step in computing the backward slice is to generate the program dependency graph. To achieve this, the program is to split into a set of nodes that represent the program statements as shown in Figure 3.

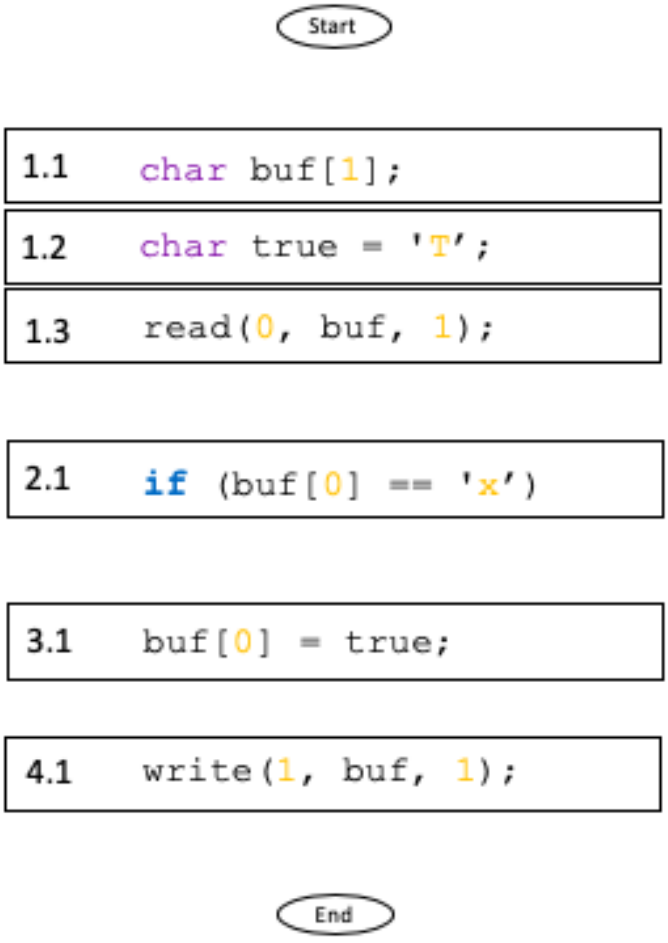

Figure 3 – Simple Example Nodes.

Next, both the data dependence and control dependence must be computed as shown in Figure 4Figure 4 – Simple Example Control and Data Dependency Graph. The data dependency is computed using the following two rules: X is found to be data dependent on Y if a variable is defined at Y and is used at X, and a path exists from Y to X where the variable is not redefined. Similarly, control dependence is determined using

the following two rules: X is found to be control dependent on Y if X is not strictly post-dominated by Y and a path exists between X and Y where every node in the path (other than X and Y) is post-dominated by Y [1]. Y post-dominates X if all possible paths from X to program exit must go through Y. In other words, if a path exists from X to exit that does not include Y, Y does not post-dominate X.

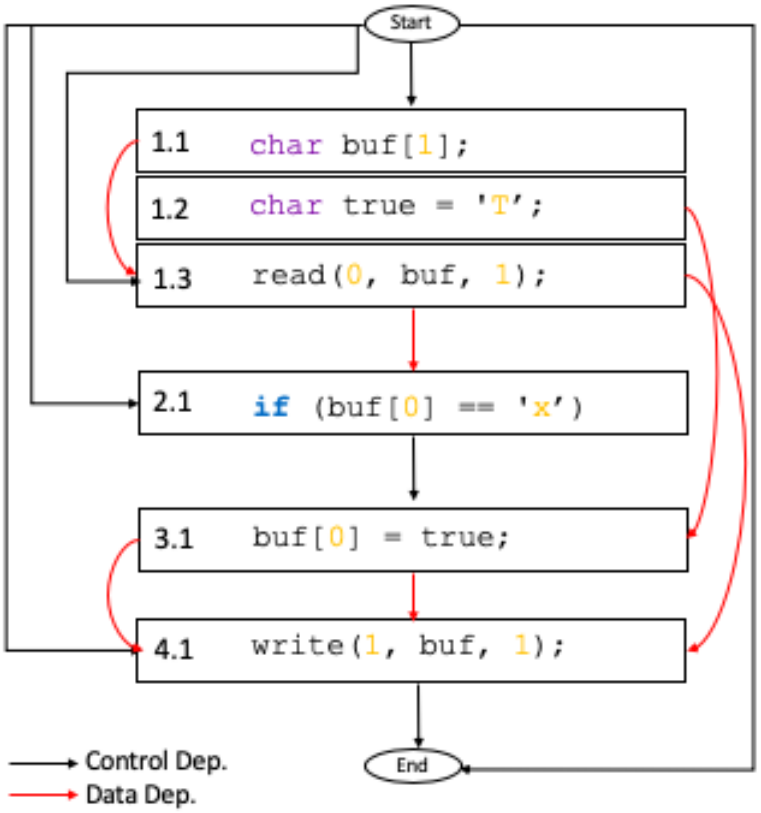

Figure 4 – Simple Example Control and Data Dependency Graph.

Given this newly formed graph, a slice can now be computed. Given the target of a backward slice, a slice is computed by navigating the PGD through the backwards reachable nodes beginning at the target [1]. See the examples below in Figure 5 and Figure 6 where the nodes highlighted in grey are in the slice [1].

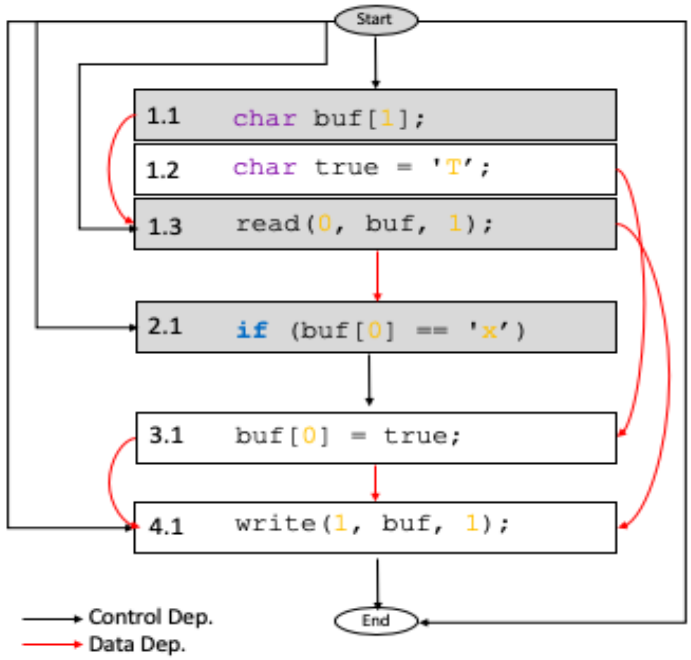

Figure 5 – Simple Example Slice at 2.1.

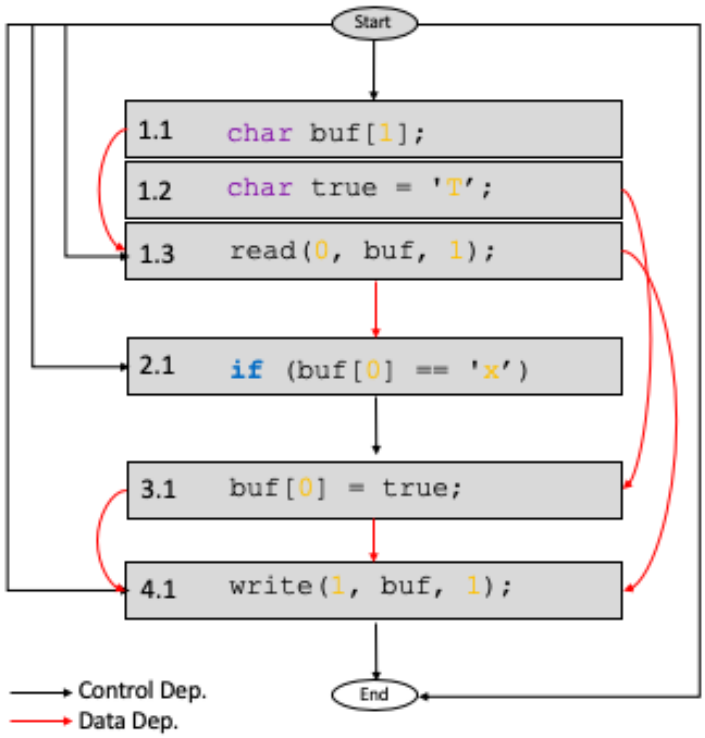

Figure 6 – Simple Example Slice at 4.1.

For all of the examples to follow, the slice being investigated is for block 4 line 1.

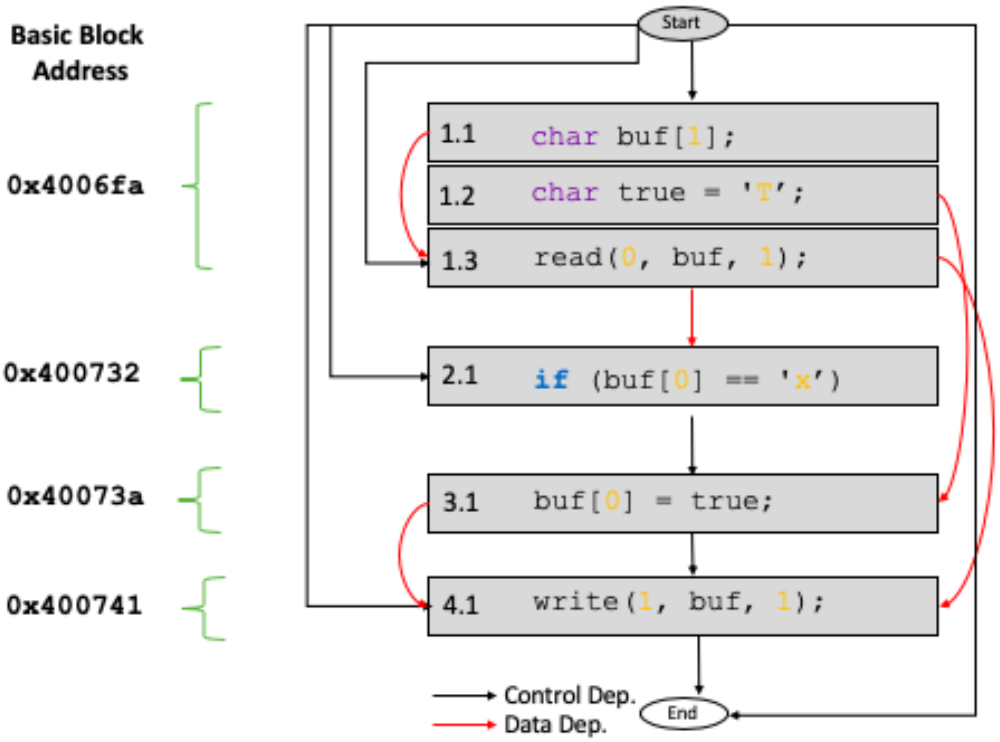

Figure 7 – Simple Example Slice at 4.1 Nodes.

The block addresses are: 0x4006af which refers to block 1, 0x400732 which refers to block 2, 0x40073a which refers to block 3, and finally 0x400741 which refers to block 4. See Figure 8 below for a graphic showing what addresses are contained in each basic block.

Figure 8 - Simple Example Basic Block Nodes Member Addresses.

The next step is to explore what instructions should be the targets of the backward slice analysis. In other words, what are the instructions that we are interested in. For this binary we are interested in the contents of the buffer when it is written. This being the case, the target instructions are chosen to be the following: 0x400752 ([write (1, buf, 1);]), 0x40074a (pointer to the buffer), and 0x40073e ([buf[0] = true;]). Graphics indicating what addresses were brought into the slice by the selected targets can be seen below in Figure 9, Figure 10, Figure 11, and Figure 12.

Figure 9 – Simple Example Addresses in the Backward Slice Brought in by 0x400752.

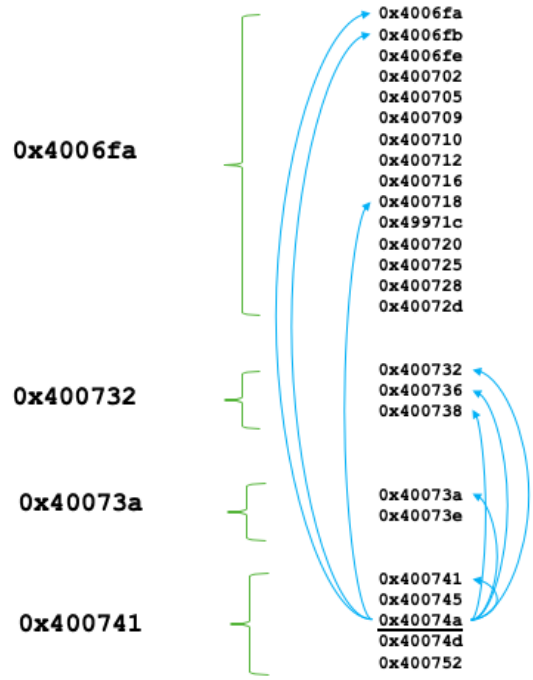

Figure 10 – Simple Example Addresses in the Backward Slice Brought in by 0x40074a.

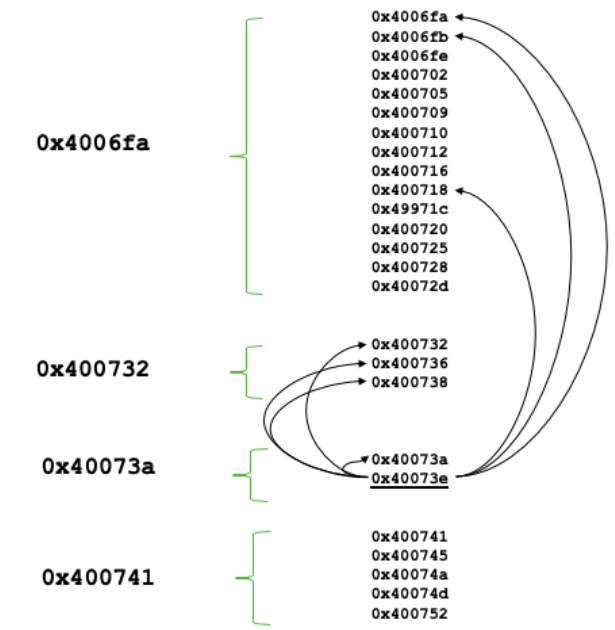

Figure 11 – Simple Example Addresses in the Backward Slice Brought in by 0x40073e.

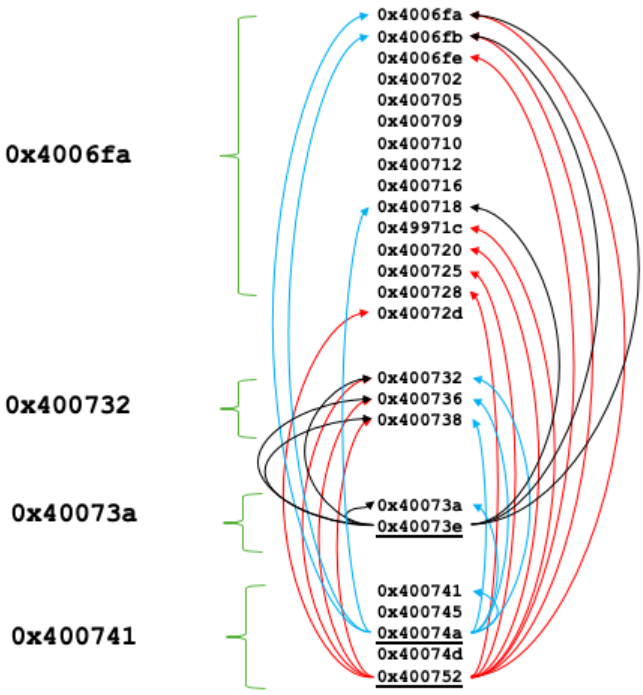

Figure 12 – Simple Example Addresses in the Backward slice Brought in by All Three Targets.

The backward slice graph shown in Figure 13 shows the addresses in the slice, what target address brought them into the slice, and how they were brought in, weather that be through data dependence or control dependence. The backward slice only includes addresses that are in the path. However, if the condition at 0x400736 is met, then a pre-defined value is moved to eax (0x73a) for eventual movement into the buffer (0x40073e).

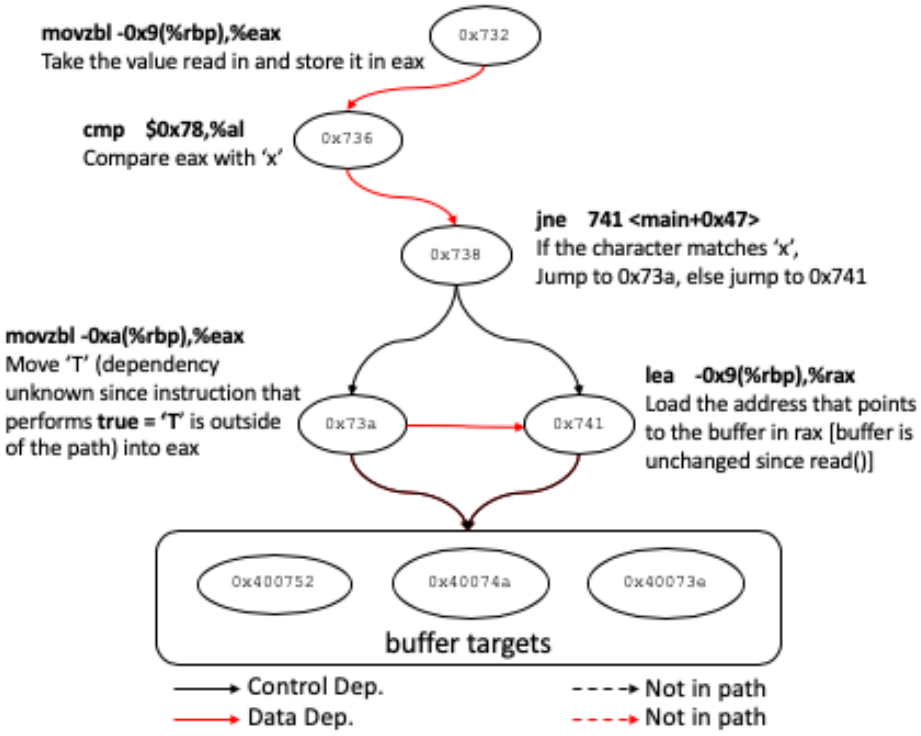

Figure 13 – Simple Example Backward Slice Hand Made Path Graph.

As shown above, there are three different ways to get to the buffer targets. These different ways to get through to the target are called path. For this example path 1 is 0x738 > 0x73a > target, path 2 is 0x738 > 0x741 > target, and path 3 is 0x738 > 0x73a > 0x741 > target . Different paths mean, potentially, different buffer contents. This being the case, it is necessary to find the information contained in the buffer for all paths to get a holistic view of what the binary can do to build the buffer contents.

The next step is to continue the backward slice past the call to read. As shown in Figure 14Figure 14 – Simple Example Backward Slice Hand Made Path Graph Past Read, the value of 'true' is set to "T" at 0x400718 highlighted In yellow. This representation of the backward slice identified dependencies for the target addresses provided.

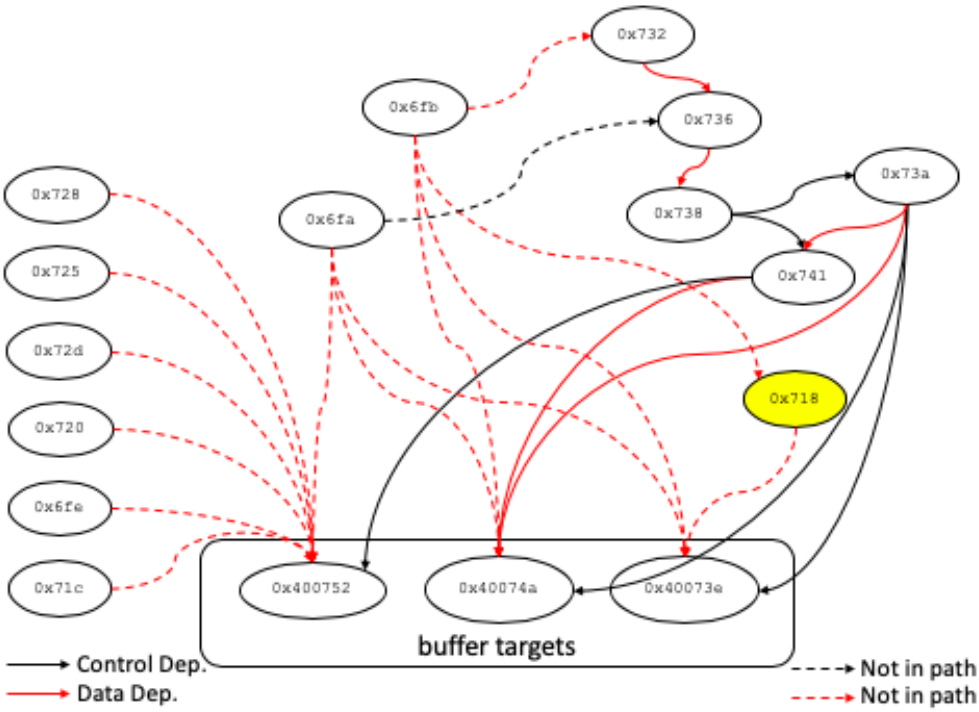

Figure 14 – Simple Example Backward Slice Hand Made Path Graph Past Read.

Based on this backward slice graph, we can see the 0x40073e target performs a memory write to the buffer. Without this target, the initialization of **true = 'T',** which is stored in the buffer, is missed.

These examples demonstrate a backward slice being computed statically. It is important to note that angr supports both static and dynamic analysis and that some of the built in functions may utilize this [3]. Additionally, this set of examples thoroughly demonstrate the ideas surrounding and the utility of backward slice analysis.

1.1.4 Automation

As demonstrated above, the process of generating useful and complete backward slice graphs is highly involved even for simple binary examples. The ability to automate the process of constructing these graphs is central to the utility of this project. With the detailed backward slice information presented in a graph as shown in Figure 14, an analyst is capable of more easily and quickly identifying what parts of a binary should be investigated to characterize malicious behaviour. For example, if an analyst is reversing a malware sample that is suspected of sending personal information of the user to an attacker, these tools are capable of demonstrating where that personal information is being collected by the binary given the address of a network call. The automation of the backward slice graph reduces the burden of this portion of the analysis while also standardizing the output to allow for more streamlined backward slice examination. The blocks that are not represented in the backward slice are definitionally not a factor in the construction of the aforementioned network call, and are therefore not worth the time of an analyst to investigate.

1.2 Tools Used

A variety of tools are employed in development and deployment of this project. This section serves as a summary of the tools utilized:

1.2.1 Python

Python is an open source programming language that is designed around the ethos of enabling developers to “work quickly and integrate systems more effectively” by simplifying the process of working with external tools [2]. This project extensively uses Python and many of the packages developed for it. This section serves as a summary of the tools utilized.

1.2.1.1 angr

Angr is a framework, written in Python, that can be used to analyze binaries [3]. “It combines both static and dynamic symbolic ("concolic") analysis, making it applicable to a variety of tasks” [3]. Angr supports the following functionality out of the box [3]:

- Control Flow Graph Recovery
- Symbolic Execution
- Automatic ROP Chain Building
- Automatic Binary Hardening
- Automatic Exploit Generations
- GUI Based Analysis

Because angr is so full featured and in active development it is chosen as the back bone of much of this project.

1.2.1.2 NetworkX

NetworkX is a package written for Python to allow for the manipulation, creation, and study of networks [4]. NetworkX support the following features natively [4]:

- Data structures that support graphs, digraphs, and multigraphs
- Multiple standard graph algorithms
- Analysis of network structures
- Graph auto generation
- Supports nodes of any type
- Supports a multitude of edge characteristics

NetworkX is used in this project to represent the interactions between basic block nodes with “dot” graphs which are a text based way to represent graph nodes and edges.

1.2.1.3 PyGraphviz

PyGraphviz is a framework that allows Python to interface with the Graphviz visualization package [5]. It allows for the creation, reading, writing, editing, and drawing of graphs from within Python [5].

1.2.2 Docker

Docker is an instrumental tool in this project. It is one of the means by which the tools are packaged for other users to take advantage of. The tools are built into a docker container to allow for easy sharing of functionality. The docker documentation describes a container as “a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another [6].” The simplicity and functionality that docker provides makes it an easy choice as a means to distribute the tools developed for this project.

1.3 Literature Review

Several papers and journal entries are consulted in the process of this research. These resources serve as inspiration as to what types of analyses are being conducted that backward slices could offer aid to. Additionally, some of these resources act as inspiration of the capabilities that are already enabled through backward slice analysis techniques. This section serves as a summary of this literature.

1.3.1 Dispatcher

Dispatcher is a paper that focuses on the importance of automated protocol reverse engineering techniques when it comes to cyber security related applications [7]. It combines techniques of multiple previous works to create a rich field analysis system that handles automatic encryption and touts robust message tracing [7]. The example of a botnet command and control protocol is used to demonstrate the necessity of an automated system to decipher protocols [7]. The authors employ backward traversal of the execution trace to deconstruct the buffer and to compute the dependency chain that leads to a particular instruction [7]. Both of these tasks demonstrate the utility of a robust set of backward slice analysis tools.

1.3.2 Rosetta

Rosetta is a study that focuses on using binary analysis to enable researchers to perform NAT rewriting and protocol replay [8]. The researchers work to gain a deep understanding of fields in network protocols, specifically dynamic fields [8]. This paper aims to solve the problem of "dynamic fields, e.g., hostnames, IP addresses, session identifiers or timestamps" needing to be modified to successfully spoof a network protocol [8]. The researchers demonstrate in this paper that they are able to identify a

variety of dynamic fields and modify the values contained in them even under circumstances where complex encoding is utilized [8].

1.3.3 Prospex

This work is a continuation of the work done for Rosetta [9]. The authors of this paper aims to enhance the results obtained in protocol inference while also reducing the need to humans to be in the loop [9]. The claim to fame of this research is the enhanced ability to "automatically inferring state Machines [9]."

1.3.4 State of the art of network protocol reverse engineering tools

This paper "presents a survey of protocol reverse engineering tools developed in the last decade" and "considers tools focusing on the inference of the format of individual messages or of the grammar of sequences of messages [10]." The researchers aimed to classify the explored tools in a way that could reveal how the tools compared in an empirical way [10]. This background research is instrumental to understanding how backward slice analysis can be best applied to further research.

1.3.5 Extracting output formats from executables

This research paper describes the development and capabilities of tools that "extracts output data formats, such as file formats and network packet formats" from a binary that is stripped of its debug symbols in an automated manner [11]. This is accomplished by the use of "Value-Set Analysis (VSA) and Aggregate Structure Identification (ASI) to annotate HFSMs" in a way that characterizes the output values of the data contained in the communications [11]. Observing the approaches taken in this paper greatly influenced the procedures employed throughout the generation of our backward slice analysis tools.

1.3.6 Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution

This paper is a derivative work of Rosetta. The researchers demonstrate the ability to detect techniques used by encryption in binary files in an automated fashion [12]. The researched posit that existing automatic protocol reverse engineering tools are hampered in their ability to properly extract protocol fields because they are missing program semantics in the traces of the networks [12]. The techniques employed and the approaches taken in this paper served as yet another source of inspiration as to why better and more comprehensive backward slice tools is a necessity.

1.4 Protocol Inference

Communication over networks using standardized and open source network protocols (Ex. HTTP, FTP, and SMTP) is commonplace for computer programs [13]. Communications that utilize these open source network protocols can be trusted because they are well understood and therefore easy to validate. However, malicious programs, such as viruses and malware, frequently use proprietary network protocols to hide their communications from the user. These protocols are generally much more complex and employ obfuscation techniques to hide the contents of their communication. Amplifying their complexity, malicious programs can be packaged in such a way that static reverse engineering by a human analyst is unreasonable.

Taking inspiration from Dispatcher, the backward slice tools developed for this project have applications in the inference of information about the network protocol that an unknown or obfuscated binary is employing [3]. This type of analysis can be split, primarily, into two distinct categories. Firstly, network-based approaches evaluate sample packets collected from the network communication of the binary. Secondly, application-based approaches analyze the program code of the binary. Both approaches try to generate a comprehensive picture of the protocol used [3]. The description of the employed protocol includes the following:

- State of Client Machine – Developing a model of the state of the client machine when the network call is made and determining how the sending or receiving of packets can manipulate this state.
- Encryption – Determining whether the packets sent over the network are encrypted or obfuscated in any way.

- Message Types – The packet types that are sent over the network.
- Message Fields – What variety of fields are encoded in each packet. For example, what are the data types, what are the lengths, how are they generated.

By taking inspiration from the work done in Dispatcher, this project aims to deploy a modular set of backward slice tools to more transparently communicate the operations of a binary to the analyst. This information enables an analyst to better understand what information a malicious piece of software is attempting to utilize. This empowers researchers to better combat bad actors in this space.

1.5 Spoofing Network Calls

Another possible application for these tools is to enable analysts to more efficiently spoof network calls. Using many of the same ideas as in "automated protocol inference" these tools can be used to collect information about the type and content of data being sent over the network. The collected data could be analyzed to classify what makes up a legitimate network call from the binary in question. Finally, this information can be used to construct convincing packets to send over the network.

CHAPTER 2. PROBLEM FORMULATION

This project aims to solve the issue of utilizing backward slice techniques to gather specific and detailed information about the activities of malicious binaries such as malware. The tools are designed to be modular to allow flexibility on how they are utilized. They are functional as a stand-alone analysis system as well as capable of being embedded in larger analysis suits. As stated above, the ideal application of these tools will enable an analyst to more quickly and easily employ backwards slice analysis to glean actionable information about the portions of a malicious binary that are responsible for a suspicious network call. Given that some information is known about the network call in question, these tools automatically present actionable information about how the contents of the call are constructed.

2.1 Goals

The overarching goal of this project is to develop analysis tools that enable an analyst to glean useful information from the backward slice of a given binary file. Specifically, this paper will need to demonstrate that the tools developed are capable of assisting an analyst in investigating how malicious binaries construct calls. This can be applied to a variety of calls, including network calls.

This project makes extensive use of angr because of the vast capabilities it has and relatively low barrier to entry. However, because the backward slice analysis built into angr is not fool proof, this project aims to build upon it and make it more user friendly. The built in angr backward slice analysis suffers from an inability to clearly articulate significant information about a binary to an analyst. This project aims to rectify this to clearly and concisely display information about the backward slice to an analyst visually while also providing them with all of the raw information necessary to reconstruct the backward slice by hand if desired.

2.2 Validation

The tools developed are validated on example binaries that are written to clearly demonstrate and validate the effectiveness in investigating how the data in a function call is built. This approach allows for more rapid prototyping during the development of the aforementioned tools by drastically simplifying the source binary while also demonstrating its effectiveness in applicable problem sets. Additionally, this validation approach allows for concrete examples to be demonstrated to the reader throughout this paper.

CHAPTER 3. EXECUTION

To achieve the goals stated above, several moving pieces had to come together. This section serves as a list of the individual analysis tools developed in support of this research project. This will simply be an explanation of the tool and what role it plays in the overarching goal of making backward slice analysis more approachable and easier to integrate into more malware reverse engineering projects such as network traffic investigations. Any of the following tools are capable of being deployed in a stand-alone application, but a fully featured application of these tools makes use of them in their entirety. Such an application empowers analysts to focus more time on the important and challenging problems and less time investigating inconsequential components of the malicious binary in question.

3.1 Functions to Support Backward Slice Graph Output

3.1.1 DataModel.py

This class houses all of the data and file manipulations utilized throughout this project. For the final implementation of the backward slice graph output this only includes code to read in information from a json file that is provided by a sponsor in the engine. However, for other analyses and ongoing work this houses many other functions including, but not limited to, processing the raw backward slice into a json file, fetching the Vex ID of the instructions of a backward slice, handling the errors in a prototyped out of path calculator, output to json for the engine, and file IO manipulations.

3.1.2 gt_driver.py

This class is the interface that an analyst uses to interact with the tools in a holistic manner. It defines the command line arguments, generates the angr project, and makes calls to each of the other tools that are necessary to run the analysis in its entirety.

3.1.3 cdgGraph.py

This class flattens the CDG into a usable format. The CDG generated by angr is stored in a manner that in non-conducive to working with the backward slice. More details about this can be found in Challenges under Types of Node.

3.1.4 ddgGraph.py

This class flattens the DDG into a usable format. The DDG generated by angr is stored in a manner that in non-conducive to working with the backward slice. More details about this can be found in Challenges under Types of Node.

3.1.5 BackwardSlice.py

This class houses all of the functionality to compute the backward slice using angr. It includes the following: the ability to determine what instructions are in the path and what instructions are out of the path, compute all of the addresses in the path, all of the addresses that contain the chosen statements (individual vex statements marked as part of the backward slice), find the statement IDs for each address (must go through all the statements in the block and find the IDs that correspond to the address target address), compute the CFG and DDG, and finally compute the nodes and runs in the slice.

3.1.6 BSPathGraph.py

This class handles all of the processing and analysis to produce the image outputs of the backward slice graph analysis. It produces a graph that contains all of the targets for each path. Additionally, it outputs all of the information necessary to properly debug or hand verify each of the outputs. This debug information includes the following: DDG and CDG graphs in images and dot files of the entire binary, a list of the addresses in the path, a list of the addresses in the slice, a list of the addresses in the slice but not in the path, and CFG and DDG for the slice to each individual target.

3.2 Additional Tools Developed

In addition to the backward slice graph output, other tools were developed alongside this project to support other functionality This section serves as a summary of those tools.

3.2.1 FunctionCFG.py

This class is for analysis using CFG generated by angr. It is used for analysis on the function map of the binary.

3.2.2 PathInfo.py

This class is used to get information about a particular path. It enables an analyst to print path information in hex, print that same information in decimal, get the calls made in each path, and get function names from a particular address.

3.2.3 SubPath.py

This class is used to find the paths between two given nodes or creates a graph of all paths between two given nodes.

3.2.4 SSValidate.py

This class is used to validate if a feasible path exists between a given source and sink. This is used to automate the generation of paths to investigate with the backward slice tools.

CHAPTER 4. RESULTS

The overarching results of this project is a combination of all of small tools developed into one finished tool that constructs a visual backward slice path graph. This graph represents the results of multiple different analysis of a binary. Some of these analyses is built into angr (such as computing the CFG), but most are built specifically for this project.

This section details the results achieved. This approach demonstrates the ideas behind each step in detail while also showcasing the ability of these tools to be utilized to glean useful information about what malicious binaries are doing. Additionally, this section details the ideal insulation and application of these tools. Finally, this section serves as a landing place for the challenges faced throughout the development of these tools. It details the problems encountered and the steps taken to resolve them.

4.1 Validation

This section serves as a walkthrough of the validation process using the example code. A small c program, shown in Figure 15, is written to validate these tools. In short, this program contains a read and a write where the contents of the write are dependent on the contents of the read. This section uses the unique combination of these attributes to thoroughly explore the backward slicing and show what useful information can be gleaned from it using the tools developed for this project.

```
static char truefalse[2] = {'t', 'f'};

int main(int argc, char** argv)
{
    char buf[4];
    truefalse[0] = 'T';
    truefalse[1] = 'F';

    read(0, buf, 1);

    if (buf[0] == 'x')
    {
        buf[0] = truefalse[0];
        buf[1] = truefalse[0];
        buf[2] = truefalse[0];
        buf[3] = truefalse[0];
    }
    else if (buf[0] == 'y')
    {
        buf[0] = truefalse[1];
        buf[1] = truefalse[1];
        buf[2] = truefalse[1];
        buf[3] = truefalse[1];
    }

    write(1, buf, 4);
}
```

Figure 15 – Example Source Code.

Again, the purpose of this example binary is to make an interesting test case for a backward slice where the path between the read and resulting write can produce a variety of differing paths.

To begin, some analysis is done by hand. The first thing to do is take a look at the compiled code. A clean way to do this is with an object dump. The relevant section of the object dump of this code can be seen below in Figure 16.

```
712:   48 89 45 f8           mov    %rax,-0x8(%rbp)
716:   31 c0                 xor    %eax,%eax
718:   c6 05 f1 08 20 00 54  movb   $0x54,0x2008f1(%rip)        # [truefalse[0] = 'T';]
71f:   c6 05 eb 08 20 00 46  movb   $0x46,0x2008eb(%rip)        # [truefalse[0] = 'F';]
726:   48 8d 45 f4           lea    -0xc(%rbp),%rax
72a:   ba 01 00 00 00        mov    $0x1,%edx                   # size of the buffer = 1
72f:   48 89 c6              mov    %rax,%rsi                   # pointer to the buffer
732:   bf 00 00 00 00        mov    $0x0,%edi                   # file descriptor = 0
737:   e8 94 fe ff ff        callq  5d0 <read@plt>
73c:   0f b6 45 f4           movzbl -0xc(%rbp),%eax
740:   3c 78                 cmp    $0x78,%al                   # if buf[0] == 'y'
742:   75 2a                 jne    76e <main+0x74>             # if not x, then jump to if buf[0] == 'y'
744:   0f b6 05 c5 08 20 00  movzbl 0x2008c5(%rip),%eax        # [buf[0] = truefalse[0];]
74b:   88 45 f4              mov    %al,-0xc(%rbp)
74e:   0f b6 05 bb 08 20 00  movzbl 0x2008bb(%rip),%eax        # [buf[1] = truefalse[0];]
755:   88 45 f5              mov    %al,-0xb(%rbp)
758:   0f b6 05 b1 08 20 00  movzbl 0x2008b1target(%rip),%eax  # [buf[2] = truefalse[0];]
75f:   88 45 f6              mov    %al,-0xa(%rbp)
762:   0f b6 05 a7 08 20 00  movzbl 0x2008a7(%rip),%eax        # [buf[3] = truefalse[0];]
769:   88 45 f7              mov    %al,-0x9(%rbp)ex
76c:   eb 30                 jmp    79e <main+0xa4>
76e:   0f b6 45 f4           movzbl -0xc(%rbp),%eax
772:   3c 79                 cmp    $0x79,%al                   # if buf[0] == 'y'
774:   75 28                 jne    79e <main+0xa4>             # if not y, then jump to write()
776:   0f b6 05 94 08 20 00  movzbl 0x200894(%rip),%eax        # [buf[0] = truefalse[1];]
77d:   88 45 f4              mov    %al,-0xc(%rbp)
780:   0f b6 05 8a 08 20 00  movzbl 0x20088a(%rip),%eax        # [buf[1] = truefalse[1];]
787:   88 45 f5              mov    %al,-0xb(%rbp)
78a:   0f b6 05 80 08 20 00  movzbl 0x200880(%rip),%eax        # [buf[2] = truefalse[1];]
791:   88 45 f6              mov    %al,-0xa(%rbp)
794:   0f b6 05 76 08 20 00  movzbl 0x200876(%rip),%eax        # [buf[3] = truefalse[1];]
79b:   88 45 f7              mov    %al,-0x9(%rbp)
79e:   48 8d 45 f4           lea    -0xc(%rbp),%rax             # get buffer address and put in rax
7a2:   ba 04 00 00 00        mov    $0x4,%edx                   # size of the buffer = 4
7a7:   48 89 c6              mov    %rax,%rsi                   # pointer to the buffer
7aa:   bf 01 00 00 00        mov    $0x1,%edi                   # file descriptor = 1
7af:   e8 fc fd ff ff        callq  5b0 <write@plt>             # [write(1, buf, 4);]
```

Figure 16 – Example Object Dump.

Armed with this information, the dependency graphs can be built. Beginning with the control dependency graph shown in Figure 17, then the data dependency graph shown in Figure 18, and finally the combination of them, of the program dependency graph shown in Figure 19.

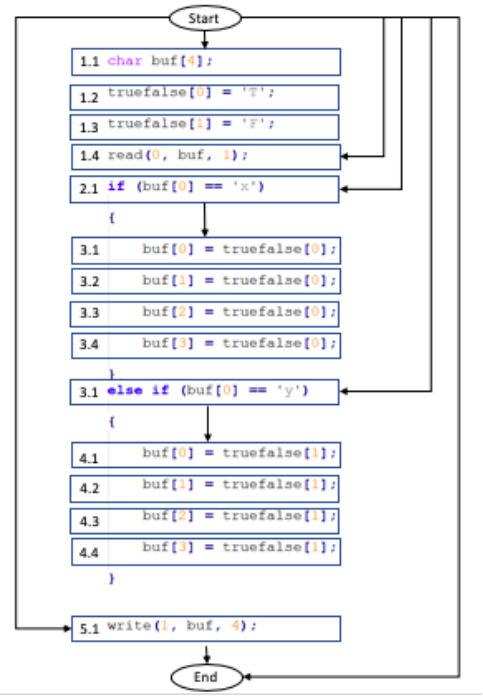

Figure 17 – Example Source Control Dependency Graph.

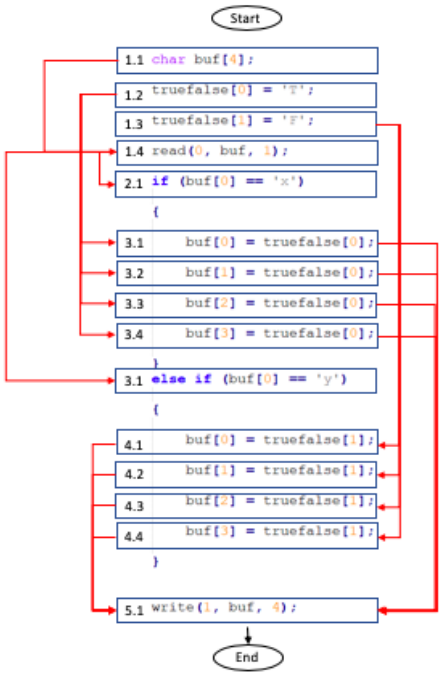

Figure 18 – Example Source Data Dependency Graph.

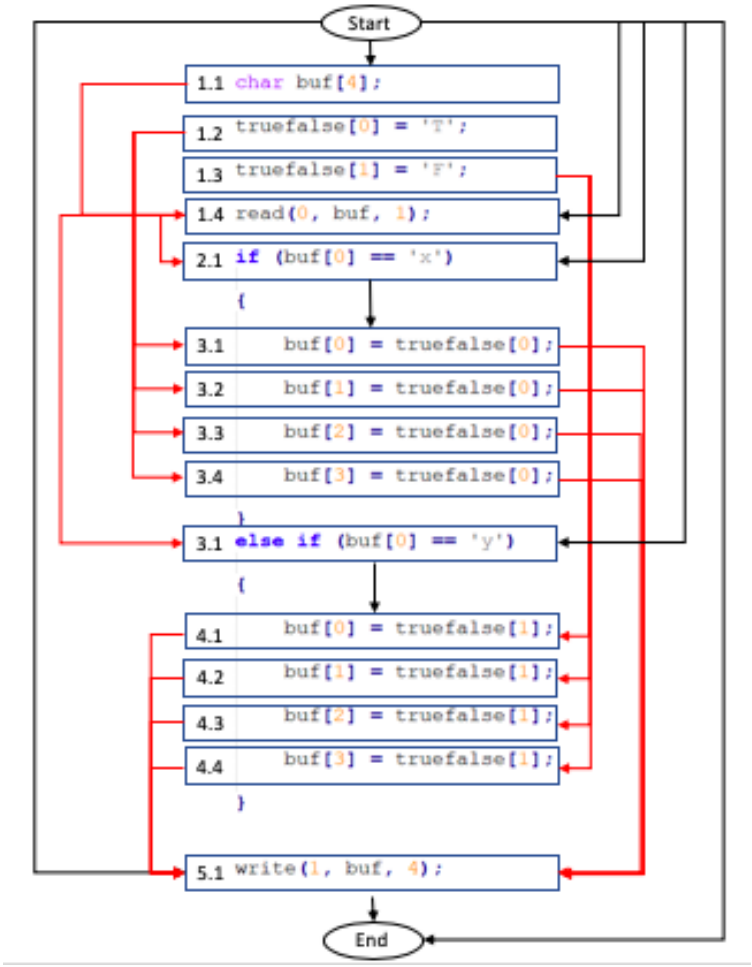

Figure 19 – Example Source Program Dependency Graph.

The PDG will be used to compute the backward slice. Angr defines a backward slice as follows: The slice of ***v*** at statement ***s*** is a set of statements used in computing ***v*** at ***s*** [3]. Unless otherwise stated, all of the following examples are for a backwards slice at 5.1. Figure 20 below describes the slice of buf at statement 5.1 is a set of statements used in computing buf at 5.1. This is path specific: buf[0] = ‘y’. As a note, the statements bordered in green are in the slice. Likewise, Figure 21 is path specific: buf[0] = ‘x’.

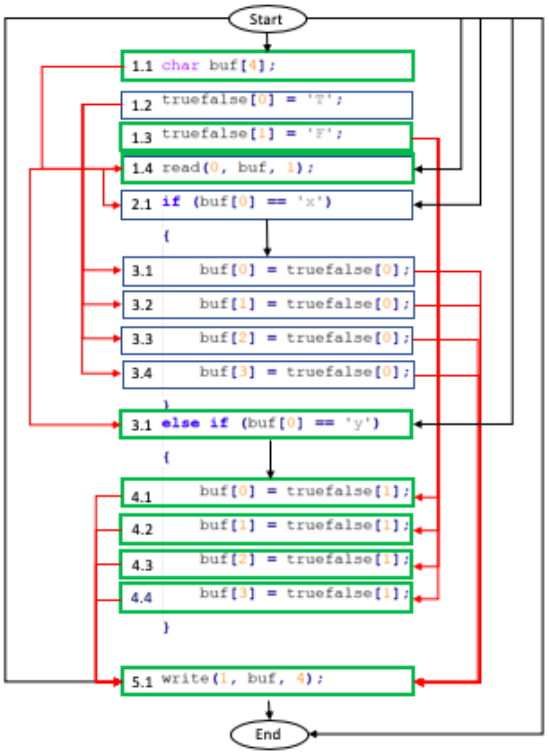

Figure 20 - Backward Slice Path 1.

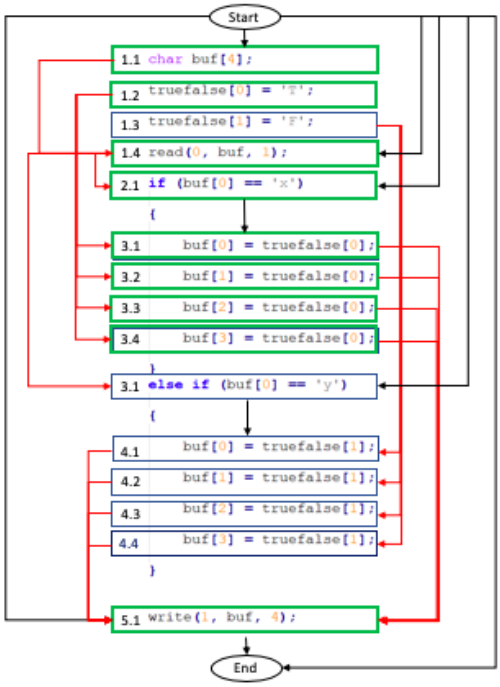

Figure 21 - Backward Slice Path 2.

Obviously, there is another path (path 3) where buf[0] != ‘x’ or ‘y’. However, this will not be explored in detail because this path does not modify the buffer. Rather, this example will include a detailed walkthrough of both paths 1 and 2. This will most clearly demonstrate the capabilities of the backward slice path graph tools. See path 1 as shown in Figure 22, and path 2 as shown in Figure 23.

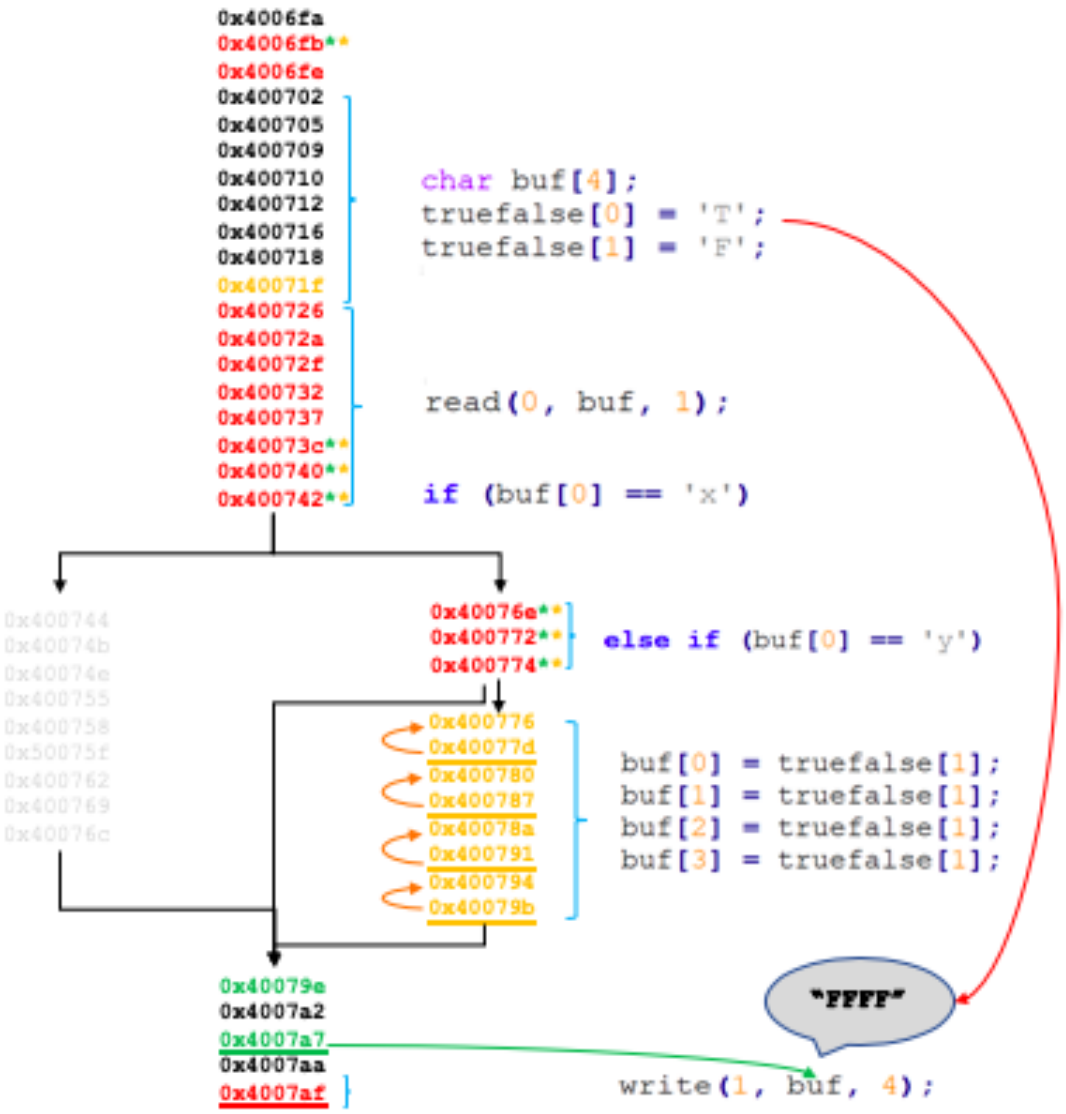

Figure 22 - Backward Slice Path 1 Details.

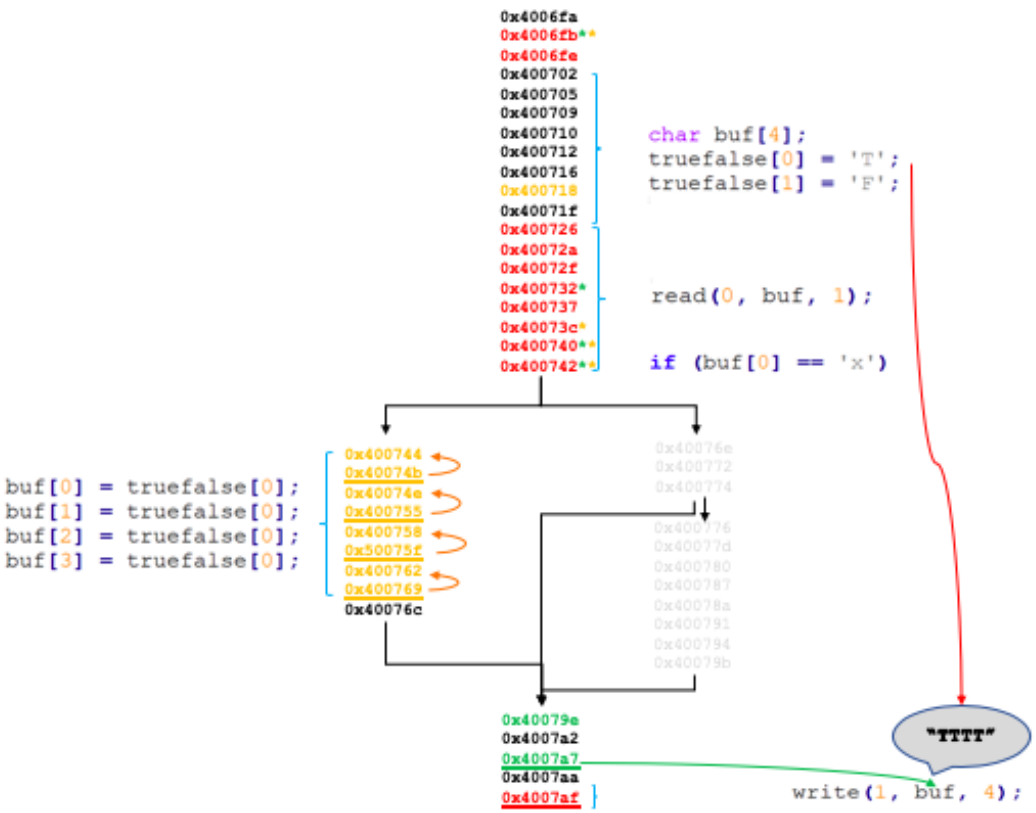

Figure 23 - Backward Slice Path 2 Details.

Figure 22 visually describes the components that make up the backward slice for path 1. The variable “truefalse” is initialized before main. However, it is later redefined at instructions 0x400718 and 0x40071f. Since the buffer is dependent upon this variable, if dependencies are incorrectly tracked, buffer resolution will incorrectly report dependencies to pre-main initializations. In this path, based on pre-main initialization, the buffer contents resolve to “ffff”. Since the target instructions for the backward slice are incomplete, the backward slice results will also be incomplete. This raises the question, what is responsible for pointing to the buffer and what is responsible for writing to the buffer? The answer to the former can be found in the second argument of the write function which is 0x4007a7. This holds the memory address of the first byte of the buffer. While adding this target instruction to the backward slice generator provides more clarity on actual dependencies, it falls short of a complete picture. Further investigation reveals instructions 0x40079b, 0x400791, 0x400787, and 0x40077d perform memory writes to bytes in the buffer. Adding these instructions to the backward slice generator finds all of the dependencies of the buffer. Based on post-main initializations, the contents of the buffer still resolve to “ffff”. As a note, the dependence of the “truefalse” variable is correctly tracked from instruction 0x40071f.

Figure 23 visually describes the components that make up the backward slice for path 2. Path 2 is similar to Path 1. Except, the path taken influences the contents of the buffer differently. In this path, based on pre-main initialization, the buffer contents resolve to “tttt”. Instructions 0x40079b, 0x400791, 0x400787, and 0x40077d perform memory writes to bytes in the buffer. Base on post-main initializations, the contents of the buffer still resolve to ‘tttt’. Again, as a note, the “truefalse” variable dependence is correctly tracked from 0x400718 much like it is from 0x40071f in path 1.

As stated above, all of the previous analysis is statically done by hand. This level of detail in highly desirable but extremely time consuming. The automation of this is the apex of this research project. At this point, angr is brought on to ease some of the analysis burden. Consequently, this is also where some angr related issues arise, as the built in backward slice functionality in angr has some robustness issues, hence the need for this project’s body of work. When these issues arise, they are identified and referenced in the section below. The first of these challenges is discussed in the section labeled as follows: “All Instructions That Modify Buffer Must Be Targets”. See these notes for a detailed explanation of the issue.

By utilizing NetworkX, the results of the backward slice can be used to track individual dependencies from the CDG and DDG. Specifically, if a dependence is found from an instruction in the backward slice to a given target, an edge is created between their node representations. The result is an automatically generated backward slice graph per path.

The first step is to use angr to generate the CFG and DDG. Use these to determine the target dependencies. For each target, search the PDG for any node that provides a dependence for the target. As an example, Target x is control dependent upon y so there is an edge from y to x. Angr is used to determine what nodes are in the given path and what nodes are not in the path. Next the determination of what addresses are and are not in the path. The outputs for both paths 1 and 2 are shown below in Figure 24 and in Figure 25.

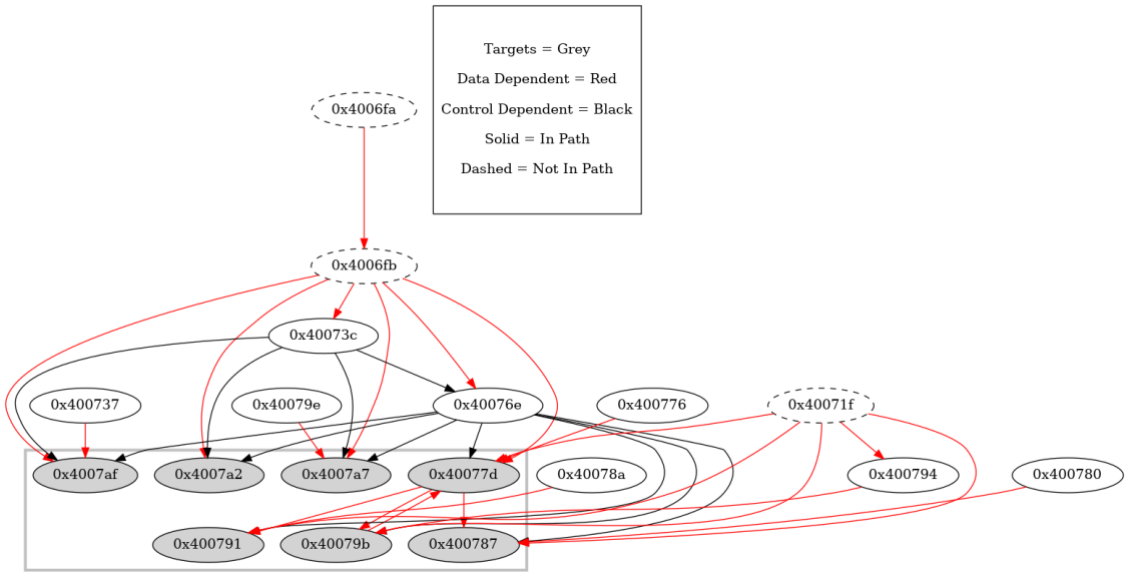

Figure 24 - Automated Backward Slice Graph Path 1.

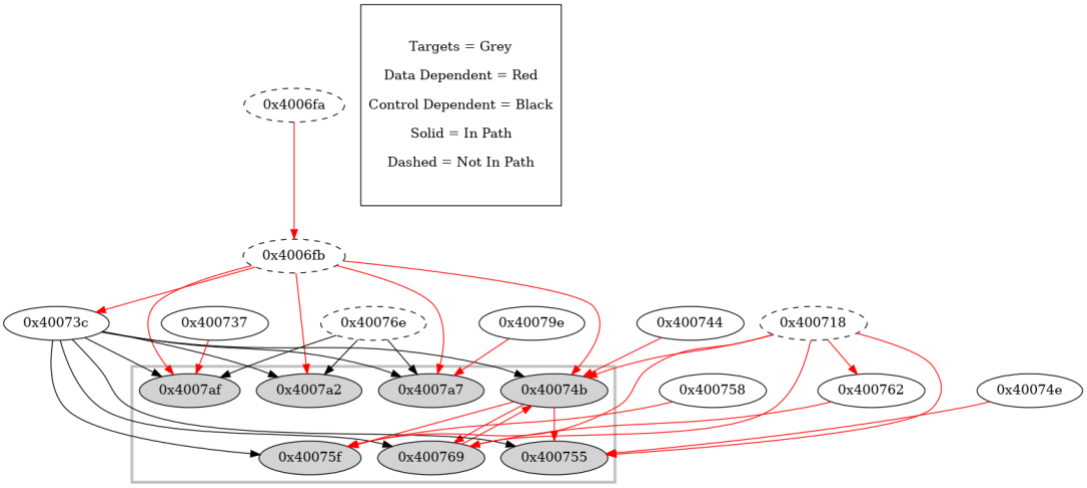

Figure 25 - Automated Backward Slice Graph Path 2.

These graphs demonstrate all of the relevant information that can be gleaned from both backward slice analysis and program dependency graphs in one simple visual. As a result, these graphs clearly articulate the chain of dependency that is responsible for each target. Armed with this information, an analyst is able to glean useful information regarding the contents of a call of interest. For example, this context would allow for the content of network call to be more closely monitored and possibly modified. Some details of the process involved in getting to this point are a bit more confusing and nuanced than make sense to explain here. See Types of Node and Visually Confusing Nodes and Edge in the section below for details.

4.2 Implementation

This section details the how the tools developed for this project can be implemented into a malware analysis workflow. It is a blueprint by which an analyst can model expectations of what these tools can bring to their reverse engineering efforts. These tools are not meant to replace the expertise required to reverse engineer malicious binaries such as malware but rather to lessen the time burden and augment the analysts abilities.

4.2.1 Integration Into Malware Analysis Workflow

The tools developed for this project have applications in a malware analysis workflow in assisting an analyst in better focusing their efforts on functions of the binary that are of the most interest. This is enabled by enumerating what blocks are responsible for constructing the contents of a particular function call. This requires that an analyst has some existing knowledge of the binary.

Given that an analyst is able to identify the address of a network call, these tools are capable of identifying where the analyst should look to understand what information the malware is collecting for this call. This information enables analysts to do the following: understand what information has been compromised on a particular system, reconstruct the command and control messages between a malware installation and its operators, and spoof the network call with feasible data to learn more about the malware operator or waste their resources.

4.3 Challenges

This section details the challenges, roadblocks, and interesting problems faced throughout the development of these tools.

4.3.1 All Instructions That Modify Buffer Must Be Targets

Because the angr addresses the backward slice, it is not sufficient to simply run the backward slice on the desired target, rather it is necessary to first compute all of the addresses that modify the buffer and tag those as targets. Some efforts have been made to automate this procedure, but this is still an open problem. Additionally, it is necessary to run a separate backward slice for each target. Otherwise, it is impossible to differentiate which instructions are responsible for bringing which target into the slice.

4.3.2 Types of Node

An quark with angr is that the CFG, DDG, and backward slice analysis represent nodes slightly differently. Some are referenced by basic block node and some by instructions. This is a technicality, and it is mostly straightforward to convert them all to the same format. This is handled by cdgGraph.py and ddgGraph.py. For reference, the backward slice graph output references by instruction as shown in Figure 24 and Figure 25.

4.3.3 Visually Confusing Nodes and Edges

An artefact left over by the way the backward slices are computed for each path is that it is sometimes impossible to tell which target is responsible for a connection. For example, if a target in the slice because of a particular node but ends up being a target for another backward slice in the same path and they both end up calling each other, it is impossible to tell which one is truly responsible. Another oddity that can arise from this

approach is a situation where an instruction is shown to be data or control dependent on itself. If the situation arises where an instruction of interest is not the starting address of a basic block, but is dependent on its basic block starting address, a node can be shown to be dependent on itself. Again, this is not an incorrect representation, but can be confusing.

4.3.4 CFG Optimization Level Issue

The optimization level for CFG generation can be easily modified. A higher optimization level allows temporary variable sharing preventing the need to define and redefine variables consistently. However, temporary variable sharing creates inaccurate dependencies. For example, instruction B is data dependent upon instruction A because A defines temporary variable 15 (t15) that is also used in B. See Figure 26 below. While reducing the optimization level disallows temporary variable sharing, there can be unintended consequences preventing backward slice generation. Presently, this issue is not resolved. Therefore, some nodes may appear dependent upon another when they only share temporary variables.

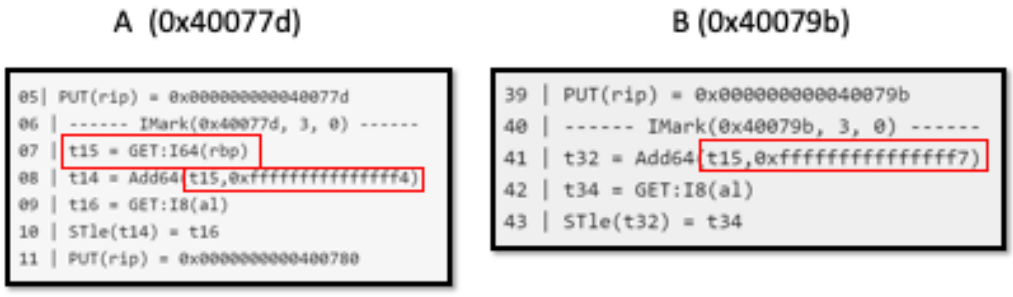

Figure 26 - CFG Optimization Level Issue.

CHAPTER 5. CONCLUSION AND FUTURE WORK

This project aims to demonstrate how backward slice analysis of binaries can be utilized to characterize the behavior of malicious software. This is accomplished in the development of tools that enable analysts to more easily and intuitively include backward slice analysis into their workflows. The ultimate purpose of the inclusion of backward slice analysis being to allow for the review of functions calls such as a network call.

To begin, the concept backward slices is clearly defined. This includes definitions, posits about when backward slice analysis is useful, and examples demonstrating the type of information that can be gleaned from them. Next the tools utilized are discussed. This includes a detailed explanation of why Python is chosen to be the primary language employed and what packages did the heavy lifting. Additionally, it is explained why docker is the obvious choice for making the tools portable. Next a deep dive into the related literature is done. This included many papers that employ backward slice analysis well and ones that would have benefited greatly from the tools developed for this project. What follows is a discussion of some applications of more powerful backward slicing tool sets. This includes protocol inference and spoofing network calls.

All of this set the ground work to demonstrate why backward slice analysis is useful in the investigation of malicious binaries. Furthermore, it allows the rest of the paper to properly articulate the role that the tools developed in this project can play in making this type of analysis more useful. The results of this research demonstrate how these tools can be utilized to enhance an analyst's ability to infer information about the network calls that a binary is making.

The work presented within this paper works toward automating the process of getting useful and human readable information out of backward slice analysis. However, the work in this area is far from being over. An automated way to generate targets that may be behaving suspiciously and the paths that are of interest would go make great strides toward making these tools more user friendly. Additionally, moving away from images as a final export of the backward slice analysis and toward an interactive GUI could make ongoing analysis much easier and more intuitive. See APPENDIX A. Development environment setup below for how to set up a development environment to expand upon these tools.

APPENDIX A. DEVELOPMENT ENVIRONMENT SETUP

Development system consists of a virtualized Ubuntu 18.04.2 LTS machine with an 8 core CPU and 16 gigabytes of ram. Analysis of larger binaries with angr necessitates such computing resources. It is not necessary to work on a virtual machine, but Ubuntu 18.04 is recommended. As per the recommendations of the angr development team, angr should be installed in a virtual environment only. However a development environment in a docker container for the suite of tools is provided.

Individual users can decide to use the provided container to construct their own Python environment to support future development. Regardless, to access both the production and development containers, simply clone the "BackwardSliceTools" repo *(git clone https://github.gatech.edu/astewart43/BackwardSliceTools.git*). For detailed instruction on how to use the provided containers, see B.1 Dockerization below. For additional information regarding the provided tools and scripts, see the github wiki page.

APPENDIX B. DOCKERIZE AND INTEGRATION

These tools are developed, in part, to support the research of other projects that the CyFI lab is involved in. Therefore, they are designed to function as stand-alone tools while also integrating seamlessly into larger tools. To that end, all of the tools developed for this project are packaged in a stand-alone docker container. This allows any user to utilize the functionality of the tools without concern for platform compatibility. Additionally, many of our backward slice functionality has been integrated into a binary analysis Engine developed by a sponsor. Details of this procedure are further discussed below.

B.1 Dockerization

In the interest of enabling the widest use of the tools developed for this project possible, a development environment and finished production environment are built into containers that are available in the github. With either case, the following steps need to be taken to build and run the containers on another system.

- Install Docker here: https://docs.docker.com/get-started/
- Clone https://github.gatech.edu/astewart43/BackwardSliceTools.git to your local machine
- Change directories into “BackwardSliceTools”
- Change directories into either “ProductionContainers” or “DevelopmentContainer”
- Follow the instructions in “README.md” found in that container.

B.2 Engine Integration

A sponsor if this research project is working on a binary analysis tool set project. The goal of this project is to develop a suite of binary analysis tools and integrate them all into the an Engine. The Engine is a command line interface that enables an analyst to perform a multitude of operations on a binary file. To integrate the tools developed for this project into the engine, the following need to be done:

- Make the output a printable dictionary in JSON format that the engine CLI will recognize. This is how the engine expects data to be stored for other functions to access results.
- Make the tools developed for this project take in variables from the engine in the form of command line arguments. For example, the engine is capable of enumerating execution paths through the binary. This information can be used while computing the backward slice.
- Enable the Python script to be callable from a shell script. This is how the engine interacts with all its external functionality.
- Make appropriate edits to the engine's code to allow the shell script to be called. This includes the following:
    - Make a function that calls the shell script, parses the output, prints it for the analyst to see, and saves the output in a global variable.
    - Add the command to the engine's list of known commands and point this command to the corresponding function.
    - Add the command to the help print menu.

REFERENCES

[1] Saltaformaggio, B "Software Vulnerabilities and Security Fall 2017, Slide Deck 9 – Program Slicing". Available at: https://gatech.instructure.com/courses/789/files/folder/slides?preview=80705

[2] The Python Software Foundation, "Python". Available at: https://www.python.org.

[3] Y. Shoshitaishvili, R. Wang, A. Dutcher, L. Dresel, E. Gustafson, N. Redini, P. Grosen, C. Unger, C. Salls, N. Stephens, C. Hauser, and J. Grosen, "angr," *angr*. Available at: http://angr.io/.

[4] NetworkX Developers, "NetworkX: Software for complex networks". Available at: https://networkx.github.io.

[5] PyGraphviz Developer Team, "PyGraphviz". Available at: http://pygraphviz.github.io

[6] Docker Inc, "Docker". Available at: https://www.docker.com.

[7] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. 2009. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In Proceedings of the 16th ACM conference on Computer and communications security, 621–634. Available at: http://bitblaze.cs.berkeley.edu/papers/dispatcher_ccs09.pdf

[8] Juan Caballero and Dawn Song. 2007. Rosetta: Extracting Protocol Semantics using Binary Analysis with Applications to Protocol Replay and NATRewriting. CyLab (2007), 32. Available at: https://software.imdea.org/~juanca/papers/cmucylab07014.pdf

[9] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. 2009. Prospex: Protocol specification extraction. In 2009 30th IEEE Symposium on Security and Privacy, 110–125. Available at: https://sites.cs.ucsb.edu/~chris/research/doc/oakland09_prospex.pdf

[10] Julien Duchêne, Colas Le Guernic, Eric Alata, Vincent Nicomette, and Mohamed Kaâniche. 2018. State of the art of network protocol reverse engineering tools. J. Comput. Virol. Hacking Tech. 14, 1 (February 2018), 53–68. Available at: https://hal.inria.fr/hal-01496958/document

[11] Junghee Lim, Thomas Reps, and Ben Liblit. 2006. Extracting output formats from executables. In 2006 13th Working Conference on Reverse Engineering, 167–178. Available at: http://pages.cs.wisc.edu/~liblit/wcre-2006/wcre-2006.pdf

[12] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. 2008. Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution. In NDSS, 1–15. Available at: https://personal.utdallas.edu/~zxl111930/file/NDSS08_AutoFormat.pdf

[13] Donghao Zhou, Zheng Yan, Yulong Fu, and Zhen Yao. 2018. A survey on network data collection. J. Netw. Comput. Appl. 116, (2018), 9–23. Available at: https://doi.org/10.1016/j.jnca.2018.05.004